Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates

by

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

and

Neil Coburn and Per-Åke Larson
Department of Computer Science
University of Waterloo
Waterloo, Ontario, N2L 3G1 Canada

TECHNICAL REPORT NO. 235

Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates

by

José A. Blakeley, Neil Coburn and Per-Åke Larson

November, 1987

To appear in *ACM Trans. on Database Systems.*

Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates*

José A. Blakeley Neil Coburn Per-Åke Larson†
University of Waterloo, Canada

Abstract

Consider a database containing not only base relations but also stored derived relations (also called materialized or concrete views). When a base relation is updated, it may also be necessary to update some of the derived relations. This paper gives sufficient and necessary conditions for detecting when an update of a base relation cannot affect a derived relation (an irrelevant update), and for detecting when a derived relation can be correctly updated using no data other than the derived relation itself and the given update operation (an autonomously computable update). The class of derived relations considered is restricted to those defined by PSJ-expressions, that is, any relational algebra expression constructed from an arbitrary number of project, select and join operations (but containing no self-joins). The class of update operations consists of insertions, deletions, and modifications, where the set of tuples to be deleted or modified is specified by a select condition on attributes of the relation being updated.

Categories and Subject Descriptions: H.2.1[**Database Management**]:Logical Design—Data Models
General Terms: Theory, Performance
Additional Key Words and Phrases: Database design, Relational databases, Conceptual relations, Stored relations, Pre-joined relations, Derived relations, Materialized views

1 Introduction

In a relational database system, the database may contain *derived relations* in addition to base relations. A derived relation is defined by a relational expression (query) over the base relations.

*Reference [6] is an early and incomplete version of this paper.

†This research was supported by Cognos, Inc., Ottawa under contract WRI 502-12, by the National Council of Science and Technology of México (CONACYT), by a Natural Sciences and Engineering Research Council of Canada (NSERC) Postgraduate Scholarship and under NSERC grant No. A-2460.

Authors' addresses: J.A. Blakeley, Computer Science Department, Indiana University, Bloomington, Indiana, 47405-4101 U.S.A.; N. Coburn and P.-Å. Larson, Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1 Canada.

A derived relation may be *virtual*, which corresponds to the traditional concept of a view, or *materialized*, meaning that the relation resulting from evaluating the expression over the current database instance is actually stored. In the sequel all derived relations are assumed to be materialized, unless stated otherwise. As base relations are modified by update operations, the derived relations may also have to be changed. A derived relation can always be brought up to date by re-evaluating the relational expression defining it, provided that the necessary base relations are available. However, complete re-evaluation of the expression is often wasteful, and the cost involved may be unacceptable.

Consider a database scheme $\mathbf{D} = (D, S)$ consisting of a set of base relation schemes $D = \{R_1, R_2, \ldots, R_m\}$ and a set of derived relation definitions $S = \{E_1, E_2, \ldots, E_n\}$, where each $E_i \in S$ is a relational algebra expression over some subset of D. Suppose that an update operation $\mathcal{U}$ is posed against the database d on D specifying an update of base relation r_u on $R_u \in D$. To keep the derived relations consistent with the base relations, those derived relations whose definition involve R_u may have to be updated as well. The general *maintenance problem for derived relations* consists of: (1) determining which derived relations may be affected by the update $\mathcal{U}$, and (2) performing the necessary updates to the affected derived relations efficiently.

As a first step towards the solution of this problem, we consider the following two important subproblems: Given an update operation $\mathcal{U}$ and a potentially affected derived relation E_i,

- determine the conditions under which the update $\mathcal{U}$ cannot have any effect on the derived relation E_i, regardless of the database instance. In this case, the update $\mathcal{U}$ is said to be *irrelevant* to E_i.

- if the update $\mathcal{U}$ is not irrelevant to E_i, then determine the conditions under which E_i can be correctly updated using only $\mathcal{U}$ and the current instance of E_i, for every instance of the database. That is, no additional data from the base relations D is required. In this case, the effect of $\mathcal{U}$ on E_i is said to be *autonomously computable.*

In this paper we give necessary and sufficient conditions for detecting irrelevant and autonomously computable updates. The class of derived relations is restricted to those defined by *PSJ*-expressions, that is, any relational algebra expression constructed from an arbitrary number of project, select, and join operations. However, multiple occurrences of the same relation in the expression are not allowed (self-joins). The class of update operations consists of insertions, deletions, and modifications where the set of tuples to be deleted or modified is specified by a select condition on the attributes of the relation being updated. We have implemented a simple prototype capable of detecting irrelevant and autonomously computable updates; some experimental results are reported in the last section of this paper. Testing the conditions eventually requires testing the satisfiability of certain Boolean expressions, which, in general, is an *NP*-complete problem. Even though we impose some restrictions on the atomic conditions from which the Boolean expressions are built, we cannot avoid the exponential growth characteristic of *NP*-complete problems. However, the exponential growth depends on the number of attributes and atomic conditions in the selection

conditions of the update operation and the derived relation. Experimental results indicate that, normally, this is not a severe problem.

The maintenance problem for derived relations is part of an ongoing project at the University of Waterloo on the use of derived relations. The project is investigating a new approach to structuring the database in a relational system at the internal level. In current systems there is, typically, a one-to-one correspondence, in terms of data contents, between conceptual relations and stored relations. (However, an implementation may map stored relations into physical files in various ways, see [3].) This is a simple and straightforward solution, but its drawback is that the processing of a query often requires data to be collected from several stored relations. Instead of directly storing each conceptual relation, we propose structuring the stored database as a set of derived relations. The choice of stored relations should be guided by the actual or anticipated query load so that frequently occurring queries can be processed rapidly. To speed up query processing, some data may be redundantly stored in several derived relations.

The structure of the stored database should be completely transparent at the user level. This requires a system capable of automatically transforming any user update against a conceptual relation, into equivalent updates against all stored relations affected. The same type of transformation is necessary to process user queries. That is, any query posed against the conceptual relations must be transformed into an equivalent query against the stored relations. The query transformation problem has been addressed in papers by Larson and Yang [12,13,18].

Although our main motivation for studying the problem stems from the above project, its solution also has applications in other areas of relational databases. Buneman and Clemons [8] proposed using views (that is, virtual derived relations) for the support of alerters. An alerter monitors the database and reports when a certain state (defined by the view associated with the alerter) has been reached. Hammer and Sarin [11] proposed a method for detecting violations of integrity constraints. Certain types of integrity constraints can be seen as defining a view. If we can show that an update operation has no effect on the view associated with an alerter or integrity constraint, then the update cannot possibly trigger the alerter or result in a database instance violating the integrity constraint. The use of derived relations (called concrete views) for the support of real-time queries was suggested by Gardarin et al. [10], but it was discarded because of the lack of an efficient update mechanism. Our results have direct application in this area.

The detection of irrelevant or autonomously computable updates also has applications in distributed databases. Suppose that a derived relation is stored at some site and that an update request, possible affecting the derived relation, is submitted at the same site. If the update is autonomously computable, then the derived relation can be correctly updated locally without requiring data from remote sites. If the request is submitted at a remote site, then we need to send only the update request itself to the site of the derived relation. As well, the results presented here provide a starting point for devising a general mechanism for database snapshot refresh [2,7,14].

2 Notation and Basic Assumptions

A *database scheme* $\mathbf{D} = (D, S)$ consists of a set of (*base*) *relation schemes* $D = \{R_1, R_2, \ldots, R_m\}$, and a set of derived relation definitions $S = \{E_1, E_2, \ldots, E_n\}$, where each $E_i \in S$ is a relational algebra expression over some subset of D. A *database instance* d, consists of a set of *relation instances* $r_1, r_2, \ldots, r_m$, one for each $R_i \in D$. We impose no constraints (e.g., keys or functional dependencies) on the relation instances allowed. A *derived relation* $v(E_i, d)$ is a relation instance resulting from the evaluation of a relational algebra expression E_i against the database d. We consider a restricted but important class of derived relations, namely those defined by a relational algebra expression constructed from any combination of project, select and join operations, called a PSJ-expression. In addition, we impose the restriction that no relation occurs as an operand more than once in the expression. In other words, a relation cannot be joined with itself (a *self-join*). We often identify a derived relation with its defining expression even though, strictly speaking, the derived relation is the result of evaluating that expression.

We state the following without proof: every valid PSJ-expression without self-joins can be transformed into an equivalent expression in a standard form consisting of a Cartesian product, followed by a selection, followed by a projection. It is easy to see this by considering the operator tree corresponding to a PSJ-expression. The standard form is obtained by first pushing all projections to the root of the tree and thereafter all selection and join conditions. From this it follows that any PSJ-expression can be written in the form $E = \pi_{\mathbf{A}}\sigma_{C}(R_{i_1} \times R_{i_2} \times \cdots \times R_{i_k})$, where $R_{i_1}, R_{i_2}, \ldots, R_{i_k}$ are relation schemes, C is a selection condition, and $\mathbf{A} = \{A_1, A_2, \ldots, A_l\}$ are the attributes of the projection. We can therefore represent any PSJ-expression by a triple $E = (\mathbf{A}, \mathbf{R}, C)$, where $\mathbf{A} = \{A_1, A_2, \ldots, A_l\}$ is called the *attribute set*, $\mathbf{R} = \{R_{i_1}, R_{i_2}, \ldots, R_{i_k}\}$ is the *relation set* or *base*, and C is a *selection condition* composed from the conditions of all the select and join operations of the relational algebra expression defining E. The attributes in $\mathbf{A}$ will often be referred to as the *visible* attributes of the derived relation. A *selection condition* is a Boolean combination of atomic (selection) conditions. The theory developed makes no other assumptions about atomic conditions than that they are functions of attributes in the relations in the base to {*true*, *false*}. However, to be able to actually test the conditions stated in the theorems, further restrictions must be imposed on the atomic conditions allowed; this is discussed further below. We also use the notation:

$\alpha(C)$ the set of all attributes appearing in condition C

$\alpha(R)$ the set of all attributes of relation R

$\alpha(\mathbf{R})$ the set of all attributes mentioned in the set of relation schemes $\mathbf{R}$ (i.e., $\bigcup_{R_i \in \mathbf{R}} \alpha(R_i)$).

The update operations considered are insertions, deletions, and modifications. Each update operation affects only one (conceptual) relation. The following notation will be used for update operations:

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

DELETE (R_u, C_D): Delete from relation r_u all tuples satisfying condition C_D, where C_D is a selection condition over $\alpha(R_u)$.

MODIFY $(R_u, C_M, \mathbf{F}_M)$: Modify all tuples in r_u that satisfy the condition C_M, where C_M is a selection condition over $\alpha(R_u)$. $\mathbf{F}_M$ is a set of expressions, each expression specifying how an attribute of r_u is to be modified.

Note that we make the assumption that all the attributes involved in the update expressions are from relation R_u. That is, both the attributes modified and the attributes from which the new values are computed, are from relation R_u. The set of expressions $\mathbf{F}_M$ of a MODIFY operation is assumed to contain an update expression for each attribute in R_u. An *update expression* is of the form $A_i := g_i(A_{i_1}, A_{i_2}, \ldots, A_{i_k})$ where $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$ are attributes in R_u and g_i is a function over $A_{i_1}, A_{i_2}, \ldots, A_{i_k}$. This function, g_i, is called the *update function* of attribute A_i. Again, the theory developed makes no other assumptions about update functions than that they are (computable) functions on the attributes in R_u. However, in practice, additional restrictions must be placed on them in order to be able to actually test the conditions.

Note that in [6] we considered a more general class of update operations where the selection condition of DELETE and MODIFY operations may involve attributes in relations other than R_u. (Autonomously computable modifications were not considered in detail in [6].) Further work revealed that the results presented in [6] do not always hold for this more general class. However, the results are valid if the selection condition involves only attributes from R_u. This is the class of update operations considered in this paper.

For simplicity, all attribute names are taken to be unique (over the set of base relations). Current systems are capable of handling only discrete and finite domains. Any such domain can be mapped onto an interval of integers, and therefore we will in the sequel treat all attributes as being defined over some interval of integers. It will often be necessary to identify exactly from which set of attributes a tuple may take its value. Let $\mathbf{A} = \{A_1, \ldots, A_k\}$ be a set of attributes. We will use the phrase, *tuple t is defined over set* $\mathbf{A}$, to describe a situation where t is a tuple defined over the attributes $A_1, \ldots, A_k$; or more simply t *is over* $\mathbf{A}$, if no confusion will arise.

Conditions are Boolean expressions built from atomic conditions and logical connectives. An atomic condition is a function from the Cartesian product of the domains of a set of attributes (variables) to the set $\{true, false\}$. The logical connectives will be denoted by "$\vee$" for OR, juxtaposition or "$\wedge$" for AND, "$\neg$" for NOT, "$\Rightarrow$" for implication, and "$\Leftrightarrow$" for equivalence. To indicate that all variables of a condition C are universally quantified we write $\forall C$, and similarly for existential quantification $\exists C$. If we need to explicitly identify which variables are quantified, we write $\forall\ X\ (C)$ or $\exists\ X\ (C)$ where X is a set of variables.

An *evaluation* of a condition is obtained by replacing all the variable names (attribute names) by values from the appropriate domains. The result is either *true* or *false*. A *partial evaluation* (or *substitution*) of a condition is obtained by replacing some of its variables by values from the appropriate domains. Let C be a condition and t a tuple over some set of attributes. The partial

evaluation of C with respect to t is denoted by $C[t]$. The result is a new condition with fewer variables.

Detecting whether an update operation is irrelevant or autonomously computable involves testing whether certain Boolean expressions are valid, or equivalently, whether related Boolean expressions are unsatisfiable.

Definition 2.1 Let $C(x_1, \ldots, x_n)$ be a Boolean expression over variables $x_1, \ldots, x_n$. C is *valid* if $\forall x_1, \ldots, x_n C(x_1, \ldots, x_n)$ is true, and C is *unsatisfiable* if $\nexists x_1, \ldots, x_n C(x_1, \ldots, x_n)$ is true, where each variable x_i ranges over its associated domain. □

In other words, a Boolean expression is valid if it always evaluates to *true*, unsatisfiable if it never evaluates to *true*, and satisfiable if it evaluates to *true* for some values of its variables. Proving the validity of a Boolean expression is equivalent to disproving the satisfiability of its complement. Proving the satisfiability of Boolean expressions is, in general, *NP*-complete. The theory presented in this paper requires the ability to test the satisfiability of Boolean expressions. Therefore, we assume that an algorithm for testing satisfiability, for the class of Boolean expressions of interest, is available. We also assume the algorithm returns a set of values and if the given expression is satisfiable then the values satisfy the expression. Since we have imposed the restriction that attributes have finite domains and we assume that any functions used are computable we are guaranteed the existence of a satisfiability testing algorithm—though it may not be efficient.

For a restricted class of Boolean expressions, polynomial algorithms exist. Rosenkrantz and Hunt [17] developed such an algorithm for conjunctive Boolean expressions. Each expression B must be of the form: $B = B_1 \wedge B_2 \wedge \cdots \wedge B_m$ where each B_i is an atomic condition. An atomic condition must be of the form $(x\ \theta\ y + c)$ or $(x\ \theta\ c)$, where $\theta \in \{=, <, \leq, >, \geq\}$, x and y are variables representing attributes, and c is a (positive or negative) constant. Variables and constants are assumed to range over the integers. The algorithm runs in $O(n^3)$ time where n is the number of distinct variables in B.

In this paper, we are interested in the case when each variable ranges over a finite *interval* of integers. For this case, Larson and Yang [12] developed an algorithm whose running time is $O(n^2)$. However, it does not handle expressions of the form $(x\ \theta\ y + c)$ where $c \neq 0$. We have developed a modified version of the algorithm by Rosenkrantz and Hunt for the case when each variable ranges over a finite interval of integers. Details of the modified algorithm are given in Appendix A.

An expression not in conjunctive form can be handled by first converting it into disjunctive normal form and then testing each disjunct separately. In the worst case, this may cause the number of atomic conditions to grow exponentially. Several of the theorems in Sections 3 and 4 will require testing the validity of expressions of the form $C_1 \Rightarrow C_2$. The implication can be eliminated by converting to the form $(\neg C_1) \vee C_2$. Similarly, expressions of the form $C_1 \Leftrightarrow C_2$ can be converted to $C_1 C_2 \vee (\neg C_1)(\neg C_2)$. Atomic conditions of the form $(x \neq y + c)$ must be converted to $(x < y + c) \vee (x > y + c)$ to satisfy the input requirements of the Rosenkrantz and Hunt algorithm; similarly, for $(x \neq c)$.

3 Irrelevant Updates

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

This section presents necessary and sufficient conditions for the detection of irrelevant updates. The conditions are given for insert, delete, and modify operations as introduced in the previous section. First we define what it means for an update to be irrelevant.

Definition 3.1 Let d be an instance on the set of relation schemes D, and let d' be the resulting instance after applying the update operation $\mathcal{U}$ to d. Let $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$ be a derived relation definition. The update operation $\mathcal{U}$ is *irrelevant* to E if $v(E, d') = v(E, d)$ for all instances d and d'. □

If the update operation $\mathcal{U}$ does not modify any of the relations over which the derived relation is defined then, obviously, $\mathcal{U}$ cannot have any effect on the derived relation. In this case $\mathcal{U}$ is said to be *trivially irrelevant* to the derived relation.

The fact that an update is not irrelevant does not imply that the update will, in fact, affect the current instance of the derived relation. However, determining whether or not it does requires accessing the data in the database.

3.1 Irrelevant insertions

An insert operation into a base relation is irrelevant to a derived relation if it causes no tuple to be inserted into the derived relation.

Theorem 3.1 *The operation INSERT(R_u, T) is irrelevant to the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$, $R_u \in \mathbf{R}$, if and only if $\mathcal{C}[t]$ is unsatisfiable for every tuple $t \in T$.*

Proof: (Sufficiency) Consider an arbitrary tuple $t \in T$. If $\mathcal{C}[t]$ is unsatisfiable, then $\mathcal{C}[t]$ will evaluate to *false* regardless of the assignment of values to the variables remaining in $\mathcal{C}[t]$. Therefore, there cannot exist any tuple defined over the Cartesian product of the relations in $\mathbf{R} - \{R_u\}$ that would combine with t to satisfy $\mathcal{C}$ and hence cause an insertion into $v(E, d)$.

(Necessity) Consider a tuple $t \in T$, and assume that $\mathcal{C}[t]$ is satisfiable. $\mathcal{C}[t]$ being satisfiable means that there exists a tuple s defined over $\alpha(\mathbf{R})$ such that $s[\alpha(R_u)] = t$, $s[A] = \mu_A$ for every attribute $A \notin \alpha(R_u) \cup \alpha(\mathcal{C})$, where μ_A is the lowest value in the domain of A, and the rest of the values $s[A]$, $A \in \alpha(\mathcal{C}) - \alpha(R_u)$ are assigned in such a way that $\mathcal{C}[s] = \mathit{true}$. The fact that $\mathcal{C}[t]$ is satisfiable guarantees the existence of values for attributes in $\alpha(\mathcal{C}) - \alpha(R_u)$. We can then construct a database instance d using s, such that the insertion of t into r_u will cause a new tuple to be inserted into the derived relation $v(E, d)$.

To construct d, we build a relation instance r_i for each relation scheme $R_i \in \mathbf{R} - \{R_u\}$. Each relation r_i contains a single tuple t_i, where $t_i = s[\alpha(R_i)]$. The database instance d consists of the relation $r_u = \emptyset$ and relations $r_i = \{t_i\}$ for each $R_i \in \mathbf{R} - \{R_u\}$. Clearly, $v(E, d) = \emptyset$. However, if we obtain d' from d by inserting tuple t into r_u, then $v(E, d')$ will contain one tuple. Therefore, the INSERT operation is not irrelevant to the derived relation. □

3.2 Irrelevant deletions

A delete operation on a base relation is irrelevant to a derived relation if none of the tuples in the derived relation will be deleted as a result of the operation.

Theorem 3.2 *The operation DELETE (R_u, C_D) is irrelevant to the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, if and only if the condition $C_D \wedge C$ is unsatisfiable.*

Proof: (Sufficiency) If $C_D \wedge C$ is unsatisfiable, then no tuple t defined over $\alpha(\mathbf{R})$ can have values such that $C_D[t]$ and $C[t]$ are simultaneously *true*. Assume that t contains values such that $C_D[t]$ is *true*, meaning that the delete operation causes the deletion of the tuple $t[\alpha(R_u)]$ from r_u. Since t cannot at the same time satisfy C, then t could not have contributed to a tuple in the derived relation. Thus the deletion of $t[\alpha(R_u)]$ from r_u will not cause any data to be deleted from the derived relation defined by E. Therefore, the delete operation is irrelevant.

(Necessity) Assume that $C_D \wedge C$ is satisfiable. Let $\alpha(C) \cup \alpha(C_D) = \{x_1, x_2, \ldots, x_l\}$. Because $C_D \wedge C$ is satisfiable, there exists a value combination $x = (x_1^0, x_2^0, \ldots, x_l^0)$ such that $C[x] \wedge C_D[x]$ is *true*. We can then construct an instance of each relation in $\mathbf{R}$ such that deleting one tuple from r_u, $R_u \in \mathbf{R}$, will indeed change the derived relation. Each instance r_j, $R_j \in \mathbf{R}$, contains one tuple t_j constructed as follows:

- if R_j contains attribute x_k, $1 \leq k \leq l$, then $t_j[x_k] = x_k^0$.
- if R_j contains an attribute y, $y \notin \{x_1, x_2, \ldots, x_l\}$, then $t_j[y] = \mu_y$, where the value μ_y is any value in the domain of y, say the lowest value in the domain.

Initially the database instance d contains the relation instances $r_i = \{t_i\}$, $R_i \in \mathbf{R}$. Hence, $v(E, d)$ will contain one tuple. Applying the delete operation to d then gives an instance d' where the tuple t_u from relation r_u has been deleted. Clearly, $v(E, d') = \emptyset$. This proves that the deletion is not irrelevant. □

Example 3.1 Consider two relation schemes $R_1(H, I, J)$ and $R_2(K, L)$, and the following derived relation and delete operation:

$$E = (\{H, L\}, \{R_1, R_2\}, (I > J)(J = K)(K > 10))$$

DELETE $(R_1, (I < 5))$.

To show that the deletion is irrelevant to the derived relation we must prove that the following condition holds:

$$\nexists\, H, I, J, K, L\, [(I > J)(J = K)(K > 10) \wedge (I < 5)].$$

Clearly, the condition holds because the condition $(I > J)(J = K)(K > 10)$ implies that $(I > 11)$, which contradicts $(I < 5)$. Hence, the delete operation is irrelevant to the derived relation. □

3.3 Irrelevant modifications

The detection of irrelevant modifications is somewhat more complicated than insertions or deletions. Consider a tuple that is to be modified. It will not affect the derived relation if one of the following conditions applies:

- it does not qualify for the derived relation, neither before nor after the modification;
- it does qualify for the derived relation both before and after the modification, but all the attributes visible in the derived relation remain unchanged.

Theorem 3.3 introduced in this section covers the two cases mentioned above, but before we state the theorem, we need some additional notation.

Consider a modify operation MODIFY $(R_u, C_M, \mathbf{F}_M)$ and a derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$. Let $\alpha(R_u) = \{A_1, A_2, \ldots, A_l\}$. As mentioned in Section 2, we will associate an update expression with every attribute in R_u, that is, $\mathbf{F}_M = \{f_{A_1}, f_{A_2}, \ldots, f_{A_l}\}$. Each update expression is of the form $f_{A_i} \equiv (A_i := g_i(A_{i_1}, A_{i_2}, \ldots, A_{i_k}))$. If an attribute A_i is not to be modified, we associate with it a *trivial update expression* of the form $f_{A_i} \equiv (A_i := A_i)$. If the attribute is assigned a fixed value c, then the corresponding update expression is $f_{A_i} \equiv (A_i := c)$. The notation $\rho(f_{A_i})$ will be used to denote the right hand side of the update expression f_{A_i}, that is, the function after the assignment operator. The notation $\alpha(\rho(f_{A_i}))$ denotes the variables mentioned in $\rho(f_{A_i})$. For example, if $f_{A_i} \equiv (A_i := A_j + c)$ then $\rho(f_{A_i}) = A_j + c$ and $\alpha(\rho(f_{A_i})) = \{A_j\}$.

By substituting every occurrence of an attribute A_i in C by $\rho(f_{A_i})$ a new condition is obtained. We will use the notation $C(\mathbf{F}_M)$ to denote the condition obtained by performing this substitution for every variable $A_i \in \alpha(R_u) \cap \alpha(C)$. Depending on the update functions allowed, $C(\mathbf{F}_M)$ may not be in the class of Boolean expressions handled by the satisfiability algorithm in Appendix A even if C is in that class.

An update expression $\rho(f_{A_i})$ may produce a value outside the domain of A_i. We make the assumption that such a modification will not be performed, that is, the entire tuple will remain unchanged. Each attribute A_i of R_u must satisfy a condition of the form $(A_i \leq U_{A_i})(A_i \geq L_{A_i})$ where L_{A_i} and U_{A_i} are the lower and upper bound, respectively, of its domain. Consequently, the updated value of A_i must satisfy the condition $(\rho(f_{A_i}) \leq U_{A_i})(\rho(f_{A_i}) \geq L_{A_i})$ and this must hold for every $A_i \in \alpha(R_u)$. The conjunction of all these conditions will be denoted by $C_B(\mathbf{F}_M)$, that is,

$$C_B(\mathbf{F}_M) \equiv \bigwedge_{A_i \in \alpha(R_u)} (\rho(f_{A_i}) \leq U_{A_i})(\rho(f_{A_i}) \geq L_{A_i})$$

The following example illustrates the notation introduced above.

Example 3.2 Consider a relation schema $R(H, I, J)$ and the following modify operation:

$$\text{MODIFY}(R, (H > 5) \wedge (I \geq J), \{H := H + 20, I := 15, J := J\}).$$

For this modify operation we have:

$$\begin{array}{lll} f_H \equiv (H := H + 20) & \rho(f_H) \equiv H + 20 & \alpha(\rho(f_H)) = \{H\} \\ f_I \equiv (I := 15) & \rho(f_I) \equiv 15 & \alpha(\rho(f_I)) = \emptyset \\ f_J \equiv (J := J) & \rho(f_J) \equiv J & \alpha(\rho(f_J)) = \{J\} \end{array}$$

$$\mathcal{C}_M \equiv (H > 5) \wedge (I \geq J).$$

If the selection condition $\mathcal{C}$ of a derived relation is $\mathcal{C} \equiv (H > 30) \wedge (I = J)$, then

$$\mathcal{C}(\mathbf{F}_M) \equiv (H + 20 > 30) \wedge (15 = J).$$

Assuming that the domains of the variables H, I, and J are given by the ranges $[0, 50]$, $[10, 100]$, and $[10, 100]$, respectively, we obtain:

$$\mathcal{C}_B(\mathbf{F}_M) \equiv (H + 20 \geq 0) \wedge (H + 20 \leq 50) \wedge (15 \geq 10) \wedge (15 \leq 100) \wedge (J \geq 10) \wedge (J \leq 100).$$

□

We make no assumptions about the types of update functions allowed. Hence, the condition $\mathcal{C}_B(\mathbf{F}_M)$ may not be in the class of Boolean expressions of interest to us. Therefore, the satisfiability algorithm we wish to use may not be able to handle this condition.

Theorem 3.3 *The operation* $\text{MODIFY}(R_u, \mathcal{C}_M, \mathbf{F}_M)$ *is irrelevant to the derived relation defined by* $E = (\mathbf{A}, \mathbf{R}, \mathcal{C}), R_u \in \mathbf{R}$, *if and only if*

$$\forall\, [\mathcal{C}_M \wedge \mathcal{C}_B(\mathbf{F}_M) \;\Rightarrow\; (\neg\mathcal{C} \wedge \neg\mathcal{C}(\mathbf{F}_M)) \vee (\mathcal{C} \wedge \mathcal{C}(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in \mathcal{I}} (A_i = \rho(f_{A_i}))))] \qquad (1)$$

where $\mathcal{I} = \mathbf{A} \cap \alpha(R_u)$.

Proof: (Sufficiency) Consider a tuple t from the base $\mathbf{R}$ such that t satisfies $\mathcal{C}_M$ and the corresponding modified tuple, denoted by t', satisfies $\mathcal{C}_B(\mathbf{F}_M)$. Because condition (1) holds for every tuple, it must also hold for t. Hence, either the first or the second disjunct of the consequent must evaluate to *true*. (They cannot both be *true* simultaneously.)

If the first disjunct is *true*, both $\mathcal{C}[t]$ and $\mathcal{C}[t']$ must be *false*. This means that neither the original tuple t, nor the modified tuple t', will contribute to the derived relation. Hence changing t to t' will not affect the derived relation.

If the second disjunct is *true*, both $\mathcal{C}[t]$ and $\mathcal{C}[t']$ must be *true*. In other words, the tuple t contributed to the derived relation and after being modified to t', it still remains in the derived

relation. The last conjunct must also be satisfied, which ensures that all attributes of R_u visible in the derived relation have the same values in t and t'. Hence the derived relation will not be affected.

(Necessity) Assume that condition (1) does not hold. That means that there exists at least one assignment of values to the attributes, i.e., a tuple t, such that the antecedent is *true* but the consequent is *false*. Denote the corresponding modified tuple by t'. Since the consequent of condition (1) is *false*, $C[t]$ and $C[t']$ cannot both be *false*; thus there are three cases to consider.

Case 1: $C[t] = \mathit{false}$ and $C[t'] = \mathit{true}$. In the same way as in the proof of Theorem 3.1, we can then construct a database instance d from t, where each relation in $\mathbf{R}$ contains a single tuple and such that the resulting derived relation is empty. For this database instance, the modification operation will produce a new instance d' where the only change is to the tuple in relation r_u. The Cartesian product of the relations in $\mathbf{R}$ then contains exactly one tuple, which agrees with t on all attributes except on the attributes changed by the update. Hence, the derived relation $v(E, d')$ will contain one tuple since $C[t'] = \mathit{true}$. This proves that the modify operation is not irrelevant to the derived relation.

Case 2: $C[t] = \mathit{true}$ and $C[t'] = \mathit{false}$. Can be proven in the same way as Case 1, with the difference that the derived relation contains originally one tuple and the modification results in a deletion of that tuple from the derived relation.

Case 3: $C[t] = \mathit{true}$, $C[t'] = \mathit{true}$ but $\bigwedge_{A_i \in \mathcal{I}}(A_i = \rho(f_{A_i}))$ is *false*, that is, $t[A_i] \neq t'[A_i]$ for some $A_i \in \mathbf{A} \cap \alpha(R_u)$. In the same way as above, we can construct an instance where each relation in $\mathbf{R}$ contains only a single tuple, and where the derived relation also contains a single tuple, both before and after the modification. However, in this case the value of attribute A_i will change as a result of performing the MODIFY operation. Since $A_i \in \mathbf{A}$, this change will be visible in the derived relation. This proves that the update is not irrelevant to the derived relation. □

The following example illustrates the theorem.

Example 3.3 Suppose the database consists of the two relations $R_1(H, I)$ and $R_2(J, K)$ where H, I, J and K all have the domain $[0, 30]$. Let the derived relation and modify operation be defined as:

$$E = (\{I, J\}, \{R_1, R_2\}, (H > 10)(I = K))$$
$$\text{MODIFY } (R_1, (H > 20), \{(H := H + 5), (I := I)\}).$$

Thus the condition given in Theorem 3.3 becomes

$$\begin{aligned} \forall \;\; & H, I, J, K \;\; [(H > 20) \wedge (H + 5 \geq 0)(H + 5 \leq 30)(I \geq 0)(I \leq 30) \\ \Rightarrow \;\; & (\neg((H > 10)(I = K))) \wedge (\neg((H + 5 > 10)(I = K))) \\ & \vee (H > 10)(I = K)(H + 5 > 10)(I = K)(I = I)] \end{aligned}$$

which can be simplified to

$$\forall\ H, I, K\ [(H > 20)(H \leq 25)(I \geq 0)(I \leq 30)$$
$$\Rightarrow\ (\neg((H > 10)(I = K))) \wedge (\neg((H > 5)(I = K)))$$
$$\vee\ (H > 10)(I = K)].$$

By inspection we see that if $I = K$, then the second term of the consequent will be satisfied whenever the antecedent is satisfied. If $I \neq K$, the first term of the consequent is always satisfied. Hence, the implication is valid and we conclude that the update is irrelevant to the derived relation. □

The idea of detecting irrelevant updates is not new. In the work by Buneman and Clemons [8], on the support of triggers and alerters, they are called readily ignorable updates and in the work by Bernstein and Blaustein [4], on the support of integrity constraints, they are called trivial tests.

Maier and Ullman [16] study updates to relation fragments. In their work a fragment may be a physical or virtual relation over a single relation scheme, defined by selection and union operators on physical or other virtual relations. A fragment f_1 is related to fragment f_2 through a *transfer predicate* β_{12}; a Boolean expression defining which tuples from f_1 also belong to f_2. When a set of tuples is (say) inserted into f_1 only those tuples which satisfy β_{12} will be transferred to f_2. Tuples not satisfying β_{12} are irrelevant to f_2.

Our work improves upon previous work in several respects: (1) the update operations we support are more general than the ones supported in any of the above related papers, (2) we provide necessary and sufficient conditions for the detection of irrelevant updates, and (3) we provide an algorithm, for actually testing these conditions, which handles a large and commonly occurring class of atomic conditions.

Abiteboul and Vianu [1] investigated transactions that preserve a different kind of constraint, namely those defined by equality generating dependencies, total tuple generating dependencies, and acyclic inclusion dependencies. Although our notion of irrelevant updates characterizes individual updates that preserve integrity constraints defined by PSJ-expressions, the idea can be extended to groups of updates and thus to transactions. In this sense, our research complements their work.

4 Autonomously Computable Updates

Throughout this section we assume that for a given update operation and derived relation the update is not irrelevant to the derived relation. We formalize this with the following statement:

Property 1 Given an update operation $\mathcal{U}$ and the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$ then $\mathcal{U}$ is not irrelevant with respect to E.

If an update operation is not irrelevant to a derived relation, then some data from the base relations may be needed to update the derived relation. An important case to consider is one in which all the data needed is contained in the derived relation itself. In other words, the new state of the derived relation can be computed solely from the derived relation definition, the current state of the derived relation, and the information contained in the update operation. We call updates of this

type *autonomously computable updates.* Within this case, two subcases can be further distinguished depending on whether the decision is *unconditional* (scheme-based) or *conditional* (instance-based).

When the decision is unconditional, the new state of the derived relation can be computed using the definition and the current instance of the derived relation, and the information contained in the update operation, *for every database instance.* When the decision is conditional, the new state of the derived relation can be computed using the definition and the current instance of the derived relation, and the information contained in the update operation, *for the current database instance* but not necessarily for other instances. In this paper we concentrate only on the study of unconditionally autonomously computable updates, hence, we will often omit the word "unconditionally". For results on conditionally autonomously computable updates the reader is referred to [5].

Definition 4.1 Consider a derived relation definition E and an update operation $\mathcal{U}$, both defined over the database scheme D. Let d denote an instance of D before applying $\mathcal{U}$ and d' the corresponding instance after applying $\mathcal{U}$.

The effect of the operation $\mathcal{U}$ on E is said to be *unconditionally autonomously computable* if there exists a function $F_{\mathcal{U},E}$ such that

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

for every database instance d. □

The important aspect of this definition is the requirement that $F_{\mathcal{U},E}$ be a *function* of the instance $v(E, d)$. In other words, if d_1 and d_2 are database instances where $v(E, d_1) = v(E, d_2)$ 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 will be used in several proofs in this section.

Lemma 4.1 Consider a derived relation definition E and an update operation $\mathcal{U}$, both defined over the database scheme D. Let d_1 and d_2 be database instances and d_1' and d_2', respectively, be the corresponding instances after applying $\mathcal{U}$. If $v(E, d_1) = v(E, d_2)$ and $v(E, d_1') \neq v(E, d_2')$ then $\mathcal{U}$ is not autonomously computable on E.

Proof: Assume that there exists a function $F_{\mathcal{U},E}$, as in Definition 4.1, such that $v(E, d') = F_{\mathcal{U},E}(v(E, d))$ for every database instance d. Now consider the instances d_1 and d_2. 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 a function) that $F_{\mathcal{U},E}(v(E, d_1)) = F_{\mathcal{U},E}(v(E, d_2))$, that is, $v(E, d_1') = v(E, d_2')$. This contradicts the conditions given and proves the lemma. □

4.1 Basic concepts

The concepts covered by the following definitions are required in the rest of this section. They were originally introduced by Larson and Yang [12].

Definition 4.2 Let C be a Boolean expression over the variables $x_1, x_2, \ldots, x_n$. The variables $x_1, \ldots, x_k$, $k \leq n$, are said to be *nonessential* in C if

$$\forall\, x_1, \ldots, x_k, x_{k+1}, \ldots, x_n, x'_1, \ldots, x'_k$$
$$[C(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n) \Leftrightarrow C(x'_1, \ldots, x'_k, x_{k+1}, \ldots, x_n)].$$

Otherwise, $x_1, \ldots, x_k$ are *essential* in C. □

A nonessential variable can be eliminated from the condition simply by replacing it with any value from its domain. This will in no way change the value of the condition. For example, the variable H is nonessential in the condition

$$(I > 5)(J = I)((H > 5) \vee (H < 10)),$$

since the condition $(H > 5) \vee (H < 10)$ will evaluate to *true* for any value assigned to variable H. Similarly H is nonessential in

$$(I > 5)(H > 5)(H \leq 5),$$

since the condition will evaluate to *false* for any value assigned to H.

Definition 4.3 Let C_0 and C_1 be Boolean expressions over the variables $x_1, x_2, \ldots, x_n$. The variables $x_1, x_2, \ldots, x_k$, $k \leq n$, are said to be *computationally nonessential* in C_0 *with respect to* C_1 if

$$\forall\, x_1, \ldots, x_k, x_{k+1}, \ldots, x_n, x'_1, \ldots, x'_k$$
$$[C_1(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n) \wedge C_1(x'_1, \ldots, x'_k, x_{k+1}, \ldots, x_n)$$
$$\Rightarrow \ (C_0(x_1, \ldots, x_k, x_{k+1}, \ldots, x_n) \Leftrightarrow C_0(x'_1, \ldots, x'_k, x_{k+1}, \ldots, x_n))].$$

Otherwise, $x_1, x_2, \ldots, x_k$ are *computationally essential* in C_0 with respect to C_1. □

The idea behind this definition is that if a set of variables $x_1, x_2, \ldots, x_k$ are computationally nonessential in C_0 with respect to C_1, then given any tuple defined over the variables $x_1, x_2, \ldots, x_n$ satisfying the condition C_1, where the variables $x_1, x_2, \ldots, x_k$ have been projected out, we can still correctly evaluate whether the tuple satisfies the condition C_0 without knowing the exact values for the missing variables $x_1, x_2, \ldots, x_k$. This is done by assigning surrogate values to the variables $x_1, x_2, \ldots, x_k$ as explained by Larson and Yang [12].

Example 4.1 Let $C_1 \equiv (H > 5)$ and $C_0 \equiv (H > 0)(I = 5)(J > 10)$. It is easy to see that if we are given a tuple $\langle i, j \rangle$ for which it is known that the full tuple $\langle h, i, j \rangle$ satisfies C_1, then we can correctly evaluate C_0. If $\langle h, i, j \rangle$ satisfies C_1 then the value h must be greater than 5, and consequently it also satisfies $(H > 0)$. Hence, we can correctly evaluate C_0 for the tuple $\langle i, j \rangle$ by assigning to H any (surrogate) value greater than 5. □

Here is a brief description of the procedure for determining surrogate values. Consider a derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C_1)$, and suppose that we want to find which tuples in $v(E, d)$ satisfy some condition C_0. For example, C_0 may be the selection condition of a DELETE operation. Since every tuple in the derived relation satisfies C_1 we are interested in the case where all variables in the set $\mathbf{S} = (\alpha(C_0) \cup \alpha(C_1)) - \mathbf{A}^+$ are computationally nonessential in C_0 with respect to C_1. Let $\mathbf{S} = \{x_1, x_2, \ldots, x_k\}$ and $\alpha(C_0) \cup \alpha(C_1) = \{x_1, x_2, \ldots, x_n\}, n \geq k$. For each $t \in v(E, d)$ surrogate values for $x_1, x_2, \ldots, x_k$ can be computed by invoking an appropriate satisfiability testing algorithm with input $C_1[t]$. For each tuple t the algorithm returns a set of values $x_1^0, x_2^0, \ldots, x_n^0$. The values $x_1^0, x_2^0, \ldots, x_k^0$ are the required surrogate values needed to evaluate C_0 on tuple t and $x_i^0 = t[x_i]$, for $k+1 \leq i \leq n$. We are, therefore, guaranteed that surrogate values for the variables $x_1, \ldots, x_k$ exist, since $t \in v(E, d)$ implies that $C_1[t]$ is satisfiable.

Definition 4.4 Let C be a Boolean expression over variables $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$. The 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

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

□

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 $t = (x_1, \ldots, x_n)$, such that the full 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 also shown by Larson and Yang [12]. It is similar to the way surrogate values are derived for computationally nonessential variables. 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 4.2 Let $C \equiv (I = H)(H > 7)(K = 5)$. It is easy to prove that I and K are uniquely determined by H and the condition C. Suppose that we are given a tuple that satisfies C but only the value of H is known. Assume that $H = 10$. Then we can immediately determine that the values of I and K must be 10 and 5, respectively. □

Definition 4.5 Let $E = (\mathbf{A}, \mathbf{R}, C)$ be a derived relation and let $\mathbf{A}_E$ be the set of all attributes in $\alpha(\mathbf{R})$ that are uniquely determined by the attributes in $\mathbf{A}$ and the condition C. Then $\mathbf{A}^+ = \mathbf{A} \cup \mathbf{A}_E$ is called the *extended attribute set* of E. □

Larson and Yang [12] proved that $\mathbf{A}^+$ is the maximal set of attributes for which values can be reconstructed for *every* tuple of E. $\mathbf{A}^+$ can easily be computed by testing, one by one, which of the attributes in $\alpha(C) - \mathbf{A}$ are uniquely determined by C and the attributes in $\mathbf{A}$. An attribute not mentioned in C cannot be uniquely determined and, thus, cannot be in $\mathbf{A}_E$.

4.2 Insertions

It should be stressed that if the update $\mathcal{U}$ on a derived relation defined by E is autonomously computable, then the update can be performed for every derived relation instance $v(E, d)$. This characterization is important primarily because of the potential cost savings realized by updating the derived relation using only the information in its current instance. The reader should keep this in mind in this section and the subsequent ones on delete and modify updates.

Consider an operation INSERT (R_u, T) where T is a set of tuples to be inserted into r_u. Let a derived relation be defined by $E = (\mathbf{A}, \mathbf{R}, C), R_u \in \mathbf{R}$. The effect of the INSERT operation[1] on the derived relation is autonomously computable if

A. for each tuple $t \in T$ we can correctly decide whether t will (regardless of the database instance) satisfy the selection condition C and hence should be inserted into the derived relation, and

B. the values for all attributes visible in the derived relation can be obtained from t only.

Note that if t could cause the insertion of more than one tuple into the derived relation, then the update is not autonomously computable. Suppose that t generates two different tuples to be inserted: t_1 and t_2. Then t_1 and t_2 must differ in at least one attribute visible in the derived relation; otherwise only one tuple would be inserted. Suppose that they differ on $A_i \in \mathbf{A}$. A_i cannot be an attribute of R_u because the exact value of every attribute in R_u is given by t. Hence, the values of A_i in t_1 and t_2 would have to be obtained from other tuples. We cannot always guarantee that the required tuples will be available in the current instance of the derived relation.

Theorem 4.1 *Consider a derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $\mathbf{R} = \{R_1, \ldots, R_m\}$, and the update* INSERT$(R_u, \{t\})$ *where E and the update operation satisfy Property 1. The effect of the insert operation on the derived relation E is autonomously computable if and only if $\mathbf{R} = \{R_u\}$.*

Proof: (Sufficiency) If $\mathbf{R} = \{R_u\}$, then all attributes required to compute the selection condition C as well as all the visible attributes $\mathbf{A}$ are contained in the new tuple t. Hence, the function $F_{\mathcal{U},E}$ required by Definition 4.1 trivially exists and we conclude that the effect of the insertion is autonomously computable.

(Necessity) If $\mathbf{R}$ includes other base relations schemes in addition to R_u, then the insertion of tuple t into r_u may affect the derived relation defined by E. Whether it does depends on the existence of tuples in relations whose schemes are in $\mathbf{R} - \{R_u\}$. We can easily construct database instances where it is necessary to access the database to verify the existence of such tuples, even for the case when $\alpha(C) \subseteq \alpha(R_u)$ and $\mathbf{A} \subseteq \alpha(R_u)$. A database instance $d_1 = \{r_1, r_2, \ldots, r_m\}$ is constructed as follows. Each relation r_i, $1 \leq i \leq m$, $i \notin \{u, j\}$, contains a single tuple t_i, and relations r_u and r_j are empty. Similarly, construct another instance d_2 in the same manner with the one exception that r_j now contains a single tuple t_j. Clearly $v(E, d_1) = v(E, d_2) = \emptyset$. Now suppose that tuple t is inserted into r_u and furthermore, assume that $C[t] = \mathit{true}$. The existence of

[1] Recall that if $R_u \notin \mathbf{R}$, then the update cannot have any effect on the derived relation.

such a tuple t is guaranteed by the fact that the INSERT is not irrelevant. Even though tuple t satisfies the selection condition of the derived relation and it contains all visible attributes, it will not create an insertion into the derived relation in instance d_1 (because relation r_j is empty) whereas it will create an insertion in d_2. Therefore, by Lemma 4.1, the update cannot be autonomously computable. □

4.3 Deletions

To handle deletions autonomously, we must be able to determine, for every tuple in the derived relation, whether or not it satisfies the delete condition. This is covered by the following theorem.

Theorem 4.2 *The effect of the operation* DELETE(R_u, C_D) *on the derived relation* $E = (\mathbf{A}, \mathbf{R}, C)$, *where* E *and the update operation satisfy Property 1, is guaranteed to be autonomously computable if and only if the attributes in*

$$[\alpha(C_D) \cup \alpha(C)] - \mathbf{A}^+$$

are computationally nonessential in C_D *with respect to* C.

Proof: (Sufficiency) If the attributes in $[\alpha(C_D) \cup \alpha(C)] - \mathbf{A}^+$ are computationally nonessential in C_D with respect to C, then we can correctly evaluate the condition C_D on every tuple in the derived relation $v(E, d)$ by assigning surrogate values to the attributes in $\alpha(C_D) - \mathbf{A}^+$. Hence, the function $F_{U,E}$ required by Definition 4.1 exists.

(Necessity) Assume that $[\alpha(C_D) \cup \alpha(C)] - \mathbf{A}^+$ contains an attribute x, and assume that x is computationally essential in C_D with respect to C. We can then construct two tuples t_1 and t_2 over the attributes in $\mathbf{A}^+ \cup \alpha(C) \cup \alpha(C_D)$ such that they both satisfy C, t_1 satisfies C_D but t_2 does not, and t_1 and t_2 agree on all attributes except attribute x. The existence of two such tuples follows from the fact that the update is not irrelevant and from the definition of computationally nonessential attributes. In the same way as in the proof of Theorem 3.2, each of t_1 and t_2 can now be extended into an instance of D, where each relation contains a single tuple. Both instances will give the same instance of the derived relation, consisting of a single tuple $t_1[\mathbf{A}]$ (or $t_2[\mathbf{A}]$). In one instance, the tuple should be deleted from the derived relation, in the other one it should not, resulting in two different (updated) instances. Hence, by Lemma 4.1, the DELETE is not autonomously computable. □

Example 4.3 Consider two relation schemes $R_1(H, I)$ and $R_2(J, K)$. Let the derived relation and the delete operation be defined as:

$$E = (\{J, K\}, \{R_1, R_2\}, (I = J)(H < 20))$$
$$\text{DELETE}(R_1, (I = 20)(H < 30))$$

For every tuple t in E we have $\mathbf{A}^+ = \{I, J, K\}$ hence the attributes in $(\alpha(C_D) \cup \alpha(C)) - \mathbf{A}^+ = \{H, I, J\} - \{I, J, K\} = \{H\}$. In order for the effect of the deletion to be autonomously computable H must be computationally nonessential in C_D with respect to C. That is, the following condition must hold:

$$\forall \quad H, I, J, K, H' \; [(I = J)(H < 20) \wedge (I = J)(H' < 20) \\ \Rightarrow \; ((I = 20)(H < 30) \Leftrightarrow (I = 20)(H' < 30))].$$

The conditions $(H < 30)$ and $(H' < 30)$ will both be *true* whenever $(H < 20)$ and $(H' < 20)$ are *true*. For any choice of values that make the antecedent *true*, we must have $J = I$. Any value taken on by the variable I will make the condition $I = 20$ either *true* or *false*, and hence the consequent will always be satisfied. Therefore, the variable H is computationally nonessential in C_D with respect to C. This guarantees that for any tuple in the derived relation we can always correctly evaluate the delete condition by assigning surrogate values to the variable H. Notice that because $I \in \mathbf{A}^+$ is uniquely determined by C and the variables $\mathbf{A}$, we must also find surrogate values for I.

To further clarify the concept of computationally nonessential, consider the following instance of the derived relation E.

$v(E, d)$:

J	K
10	15
20	25

We now have to determine on a tuple by tuple basis which tuples in the derived relation should be deleted. Consider tuple $t_1 = \langle 10, 15 \rangle$ and the condition $C \equiv (I = J)(H < 20)$. We substitute for the variables J and K in C the values 10 and 15, respectively, to obtain $C[t_1] \equiv (I = 10)(H < 20)$. Any values for H and I that make $C[t_1] = \mathit{true}$, are valid surrogate values. For I the only value that can be assigned is 10 and for H we can assign, for example, the value 19. We can then evaluate C_D using these surrogate values, and find that $(10 = 20)(19 < 30) = \mathit{false}$. Therefore, tuple t_1 should not be deleted from $v(E, d)$. Similarly, for $t_2 = \langle 20, 25 \rangle$ we obtain $C[t_2] \equiv (I = 20)(H < 20)$. Surrogate values for H and I that make $C[t_2] = \mathit{true}$ are $I = 20, H = 19$. We then evaluate C_D using these surrogate values and find that $(20 = 20)(19 < 30) = \mathit{true}$. Therefore, tuple t_2 should be deleted from $v(E, d)$. □

4.4 Modifications

Deciding whether modifications can be performed autonomously is more complicated than deciding whether insertions or deletions can. In general, a modify operation may generate insertions into, deletions from, and modifications of existing tuples in the derived relation. In the next three sections we will state necessary and sufficient conditions for determining when a MODIFY update is autonomously computable. In Section 4.4.1 we characterize what may happen to tuples which are not in the current instance of a given derived relation; in Section 4.4.2 to tuples which are in the current instance. These two sections present conditions which are necessary for a MODIFY to be autonomously computable; in Section 4.4.3 we show that those same conditions are, collectively, also sufficient. Intuitively, the procedure required to decide whether a MODIFY is autonomously computable consists of the following steps:

A. Prove that every tuple selected for modification which does not satisfy C before modification, will not satisfy C after modification. This means that no new tuples will be inserted into the derived relation.

B. Prove that we can correctly select which tuples in the derived relation should be modified. Call this the *modify set.*

C. Prove that we can correctly select which tuples in the modify set will not satisfy C after modification and hence can be deleted from the derived relation.

D. Prove that, for every tuple in the modify set which will not be deleted, we can (autonomously) compute the new values for all attributes in **A**.

To help comprehend the subsequent discussion the reader should keep these steps in mind.

4.4.1 Tuples Outside the Derived Relation

In this section we investigate the possible outcomes for a tuple which is not in the current instance of a given derived relation. Let the derived relation of interest be defined by $E = (\mathbf{A}, \mathbf{R}, C)$ and the update by $\mathcal{U} = \text{MODIFY}(R_u, C_M, \mathbf{F}_M)$. We consider the possible outcomes of evaluating the conditions $C_M \wedge C_B(\mathbf{F}_M)$ and $C(\mathbf{F}_M)$ for a tuple t, defined over set $\alpha(\mathbf{R})$. The outcomes are given in Table 1; for completeness we include $C[t]$ even though it is never satisfied. Let us consider each

$C[t]$	$C_M[t] \wedge C_B(\mathbf{F}_M)[t]$	$C(\mathbf{F}_M)[t]$	Comments
false	*false*	*false*	No change
false	*false*	*true*	No change
false	*true*	*false*	No change
false	*true*	*true*	Insert t

Table 1: Possible results for tuples not in $v(E, d)$.

line of the table. If $C_M[t] \wedge C_B(\mathbf{F}_M)[t]$ is *false* then, t is not modified and obviously cannot cause any change in the instance of E. Note that, in this case, the value of $C(\mathbf{F}_M)$ is immaterial. This explains the first and second lines. If $C_M[t] \wedge C_B(\mathbf{F}_M)[t]$ is *true* then, since $C[t]$ is *false*, whether or not t requires a change in the instance of E depends on whether or not $C(\mathbf{F}_M)[t]$ is satisfied. That is, on whether t satisfies C after it is modified. Intuitively, if a new tuple should enter $v(E, d)$ we may not be able to determine the appropriate values for that tuple from $v(E, d)$. That is, we may need to obtain values from elsewhere in the database. Hence, to guarantee that $\mathcal{U}$ is autonomously computable we must guarantee that no tuples will be inserted into $v(E, d)$. This is the intention of the following property and subsequent theorem.

Property 2 Given the update operation MODIFY$(R_u, C_M, \mathbf{F}_M)$ and the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, the following implication is valid,

$$\forall\, (\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow \neg C(\mathbf{F}_M)).$$

Example 4.4 Suppose a database consists of the relation scheme $R(H, I)$ where H and I each have the domain [0, 50]. Let the derived relation and modify operation be defined as:

$$E = (\{H, I\}, \{R\}, (H = 20)(I < 10))$$
$$\text{MODIFY}(R, (I < 30), \{(H := H), (I := I + 1)\}).$$

The condition stated in Property 2 is then

$$\forall\, H, I$$
$$(\neg((H = 20)(I < 10))) \wedge (I < 30) \wedge (H \geq 0)(H \leq 50)(I + 1 \geq 0)(I + 1 \leq 50)$$
$$\Rightarrow \neg((H = 20)(I + 1 < 10))$$

which can be written as

$$\forall\, H, I$$
$$((H \neq 20) \vee (I \geq 10)) \wedge (I < 30) \wedge (H \geq 0)(H \leq 50)(I \geq -1)(I \leq 49)$$
$$\Rightarrow ((H \neq 20) \vee (I \geq 9)).$$

The first two atomic conditions in the antecedent are sufficient to guarantee that the consequent will evaluate to *true*. Hence, the property is satisfied for this update and derived relation and we are guaranteed that a tuple outside E will not enter E due to this update. To see why, consider the condition $(H = 20)(I < 10)$ used to select tuples for the derived relation. A tuple which does not satisfy $H = 20$ before $\mathcal{U}$ is applied will still not satisfy it after, since the value of H is not modified by $\mathcal{U}$. Similarly, since $\mathcal{U}$ *increases* the value of I, a tuple which does not satisfy $I < 10$ originally will not satisfy it after modification. □

Theorem 4.3 *If the operation* MODIFY$(R_u, C_M, \mathbf{F}_M)$ *is autonomously computable with respect to the derived relation defined by* $E = (\mathbf{A}, \mathbf{R}, C)$, *where* E *and the update operation satisfy Property 1, then Property 2 must be satisfied.*

Proof: Assume that Property 2 is not valid, that is, $[\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge C(\mathbf{F}_M)]$ is satisfiable. Let $\alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M)) = \{x_1, x_2, \ldots, x_k\}$. Because $[\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge C(\mathbf{F}_M)]$ is satisfiable, there exists a value combination $\boldsymbol{x} = (x_1^0, x_2^0, \ldots, x_k^0)$ such that $(\neg C[\boldsymbol{x}] \wedge C_M[\boldsymbol{x}] \wedge C_B(\mathbf{F}_M)[\boldsymbol{x}] \wedge C(\mathbf{F}_M)[\boldsymbol{x}])$ is *true*. We can then construct an instance of each relation in $\mathbf{R}$ such that modifying one tuple in r_u, $R_u \in \mathbf{R}$, will cause a new tuple to be inserted into the derived relation. Each instance r_j, $R_j \in \mathbf{R}$, contains one tuple t_j constructed as follows:

- if R_j contains attribute x_i, $1 \leq i \leq k$, then $t_j[x_i] = x_i^0$.

- if R_j contains an attribute y, $y \notin \{x_1, x_2, \ldots, x_k\}$, then $t_j[y] = \mu_y$, where the value μ_y is any value in the domain of y, say the lowest value in the domain.

Initially the database instance d_1 contains the relation instances $r_i = \{t_i\}$, $R_i \in \mathbf{R}$; and since $\neg C[x]$ is *true* we know that $v(E, d_1)$ is empty. Applying the modify operation to d_1 then gives an instance d_1' where the tuple t_u from relation r_u has been modified, since $C_M[x] \wedge C_B(\mathbf{F}_M)[x]$ is *true*. However, now $v(E, d_1')$ contains one tuple since $C(\mathbf{F}_M)[x]$ is *true*. We construct a second database instance d_2 from d_1 where all relation instances are the same, except instance r_u which is empty. Now, $v(E, d_2)$ is empty and $v(E, d_2')$ is empty. Hence, by Lemma 4.1 the MODIFY cannot be autonomously computable. □

4.4.2 Tuples Inside the Derived Relation

In this section we investigate the possible outcomes, under a MODIFY operation, for a tuple which is in the current instance of a given derived relation. Let the derived relation of interest be defined by $E = (\mathbf{A}, \mathbf{R}, C)$ and the update by $\mathcal{U} = \text{MODIFY}(R_u, C_M, \mathbf{F}_M)$. We again consider the possible outcomes of evaluating the conditions $C_M \wedge C_B(\mathbf{F}_M)$ and $C(\mathbf{F}_M)$ for tuple t, defined over $\alpha(\mathbf{R})$; for completeness we include $C[t]$ in Table 2. Again we consider each line of the table.

$C[t]$	$C_M[t] \wedge C_B(\mathbf{F}_M)[t]$	$C(\mathbf{F}_M)[t]$	Comments
true	*false*	*false*	No change
true	*false*	*true*	No change
true	*true*	*false*	Delete t
true	*true*	*true*	Modify t

Table 2: Possible results for tuples in $v(E, d)$.

If $C_M[t] \wedge C_B(\mathbf{F}_M)[t]$ is *false* then t is not modified and obviously cannot cause any change in the instance of E. This situation is depicted by the first two lines of the table. Note that, in this case, the value of $C(\mathbf{F}_M)$ is immaterial. Since t is already visible in $v(E, d)$ we need to be able to identify it as a tuple which will be unaffected by the update. Hence, it appears that we only need to distinguish, within $v(E, d)$, those tuples which are characterized by line one or two, and those characterized by line three or four. That is, it seems we need to evaluate $C_M \wedge C_B(\mathbf{F}_M)$ for each tuple in $v(E, d)$. This requires that all the attributes in $(\alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))) - \mathbf{A}^+$ be computationally nonessential in $C_M \wedge C_B(\mathbf{F}_M)$ with respect to C. However, as the following example illustrates, this is a slightly stronger condition than is necessary.

Example 4.5 Suppose a database consists of the relation scheme $R(H, I, J)$ where H, I, and J each have the domain [0, 50]. Let the derived relation and modify operation be defined as:

$E = (\{H\}, \{R\}, ((H = I)(H < 30)(J < 20)) \vee (H > 40))$
MODIFY$(R, (I < 40), \{(H := H), (I := I), (J := J + 5)\})$.

Therefore, the set $(\alpha(C) \cup \alpha(C_M) \cup \alpha(C_M(\mathbf{F}_M))) - \mathbf{A}^+$ is $\{I, J\}$. The test to determine if $\{I, J\}$ is computationally nonessential in $C_M \wedge C_B(\mathbf{F}_M)$ with respect to C is:

$$\begin{aligned}
&\forall\, H, I, J, I', J' \\
&\quad [(((H = I)(H < 30)(J < 20)) \vee (H > 40)) \wedge (((H = I')(H < 30)(J' < 20)) \vee (H > 40))] \\
&\quad \Rightarrow [((I < 40) \wedge (H \geq 0)(H \leq 50)(I \geq 0)(I \leq 50)(J + 5 \geq 0)(J + 5 \leq 50)) \\
&\qquad \Leftrightarrow ((I' < 40) \wedge (H \geq 0)(H \leq 50)(I' \geq 0)(I' \leq 50)(J' + 5 \geq 0)(J' + 5 \leq 50))]
\end{aligned}$$

which can be simplified to

$$\begin{aligned}
&\forall\, H, I, J, I', J' \\
&\quad [((H = I)(H < 30)(J < 20)(H = I')(J' < 20)) \vee (H > 40)] \\
&\quad \Rightarrow [((I < 40) \wedge (H \geq 0)(H \leq 50)(I \geq 0)(I \leq 50)(J + 5 \geq 0)(J + 5 \leq 50)) \\
&\qquad \Leftrightarrow ((I' < 40) \wedge (H \geq 0)(H \leq 50)(I' \geq 0)(I' \leq 50)(J' + 5 \geq 0)(J' + 5 \leq 50))].
\end{aligned}$$

If the term $((H = I)(H < 30)(J < 20)(H = I')(J' < 20))$ in the antecedent is *true* then we know $I = H = I'$ and both J and J' are less than 20, hence the consequent is *true*. On the other hand, if this term is not satisfied but $(H > 40)$ is satisfied then we know nothing about the values of I, I', J and J' and hence the consequent may not be satisfied. Therefore, we conclude that $\{I, J\}$ is computationally essential in $C_M \wedge C_B(\mathbf{F}_M)$ with respect to C.

Consider a particular tuple s over $\mathbf{R}$ where $s[\mathbf{A}] = t$ for some t in E. Since $H \in \mathbf{A}$ then we know the value of $s[H]$. If $s[H] < 30$ then we know that $s[I] = s[H]$ and $s[J] < 20$ and therefore we can evaluate $C_M \wedge C_B(\mathbf{F}_M)$ for tuple s and hence for t. The reason I is computationally essential in C_M is because for $s[H] > 40$ we do not know the value of $s[I]$ and hence cannot evaluate C_M. However, consider what would happen even if s should be modified, say to s'. The value of J would change, that is $s[J] \neq s'[J]$, but this value is neither visible in E nor used in the term $(H > 40)$. As this term would still be satisfied then s' would remain in the new instance of E and $s[\mathbf{A}] = s'[\mathbf{A}] = t$. In other words, for tuples which satisfy $(H > 40)$, it does not matter whether or not they are modified; in either case they will remain in the instance with no visible changes. Hence, the fact that we cannot evaluate C_M for these tuples does not impair our ability to determine the new instance from the current one. □

In terms of Table 2 the tuples which create this situation are some of those characterized by the fourth line. Simply because t is chosen for modification does not mean it will be *visibly* changed. In other words, if we can prove that even if t is modified it will remain in $v(E, d')$ with all the same attribute values as it had in $v(E, d)$ then it does not matter whether $C_M[t] \wedge C_B(\mathbf{F}_M)[t]$ is satisfied or not. Hence, we only need to evaluate $C_M \wedge C_B(\mathbf{F}_M)$ for those tuples which may be visibly modified. Property 3 and the theorem which follows it are intended to provide a procedure which will enable us to do this.

Property 3 Given the update operation MODIFY$(R_u, \mathcal{C}_M, \mathbf{F}_M)$ and the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$, $R_u \in \mathbf{R}$, the following condition holds

$$\forall\, t_1, t_2$$
$$[((\mathcal{C}[t_1] \wedge \mathcal{C}_M[t_1] \wedge \mathcal{C}_B(\mathbf{F}_M)[t_1]) \Rightarrow (\mathcal{C}(\mathbf{F}_M)[t_1] \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})} (A_i = \rho(f_{A_i}))[t_1])))$$
$$\vee((\mathcal{C}[t_1] \wedge \mathcal{C}[t_2]) \Rightarrow ((\mathcal{C}_M[t_1] \wedge \mathcal{C}_B(\mathbf{F}_M)[t_1]) \Leftrightarrow (\mathcal{C}_M[t_2] \wedge \mathcal{C}_B(\mathbf{F}_M)[t_2])))].$$

for t_1, t_2 over set $\mathbf{R}$ and $t_1[\mathbf{A}^+] = t_2[\mathbf{A}^+]$.

Theorem 4.4 *If the operation* MODIFY$(R_u, \mathcal{C}_M, \mathbf{F}_M)$ *is autonomously computable with respect to the derived relation defined by* $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$, *where* E *and the update operation satisfy Property 1, then Property 3 must be satisfied.*

Proof: Assume that neither term of the condition in Property 3 is satisfied. This means that for some tuple t_1 we cannot guarantee that if t_1 is in E and is modified then it will have no visible changes nor can we guarantee that $t_1[\mathbf{A}^+]$ will contain all the attribute values required to evaluate $\mathcal{C}_M \wedge \mathcal{C}_B(\mathbf{F}_M)$. Hence, there exist two tuples t_1 and t_2 over the attributes in $\mathbf{R}$ such that they both satisfy $\mathcal{C}$, t_1 satisfies $\mathcal{C}_M \wedge \mathcal{C}_B(\mathbf{F}_M)$ but t_2 does not, t_1 satisfies $\neg\mathcal{C}(\mathbf{F}_M)$ or $\neg[\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))]$, and t_1 and t_2 agree on all attributes in $\mathbf{A}^+$. Each of t_1 and t_2 can now be extended into an instance of D, d_1 and d_2 respectively, where each relation contains a single tuple. Both instances will give the same instance of the derived relation, consisting of a single tuple $t_1[\mathbf{A}] = t_2[\mathbf{A}]$. In d_1 the tuple in the derived relation should be modified. Hence, the tuple $t_1[\mathbf{A}]$ will either be deleted or a change will be made to some visible attribute; depending on whether t_1 satisfies $\neg\mathcal{C}(\mathbf{F}_M)$ or $\neg[\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))]$. In either case $v(E, d_1) \neq v(E, d_1')$. On the other hand, in the instance obtained from t_2 the tuple in the derived relation should not be modified; so $v(E, d_2) = v(E, d_2')$. Therefore, by Lemma 4.1, $\mathcal{U}$ is not autonomously computable. □

Returning to Table 2, the previous property and theorem give us a necessary condition for determining which tuples in $v(E, d)$ should be placed in the modify set and which should not. That is, for distinguishing tuples which will not be visibly changed, and hence remain in $v(E, d')$, from those that should be placed in the modify set. The next property and theorem give a necessary condition for distinguishing between those tuples in the modify set which satisfy line three and those which satisfy line four. That is, we need to determine which tuples will not still satisfy the selection condition of the derived relation after modification and should be deleted.

Property 4 Given the update operation MODIFY$(R_u, \mathcal{C}_M, \mathbf{F}_M)$ and the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, \mathcal{C})$, $R_u \in \mathbf{R}$, the attributes in the set[2] $[\alpha(\mathcal{C}) \cup \alpha(\mathcal{C}_M) \cup \alpha(\mathcal{C}_B(\mathbf{F}_M))] - \mathbf{A}^+$ are computationally nonessential in $\mathcal{C}(\mathbf{F}_M)$ with respect to the condition $\mathcal{C} \wedge \mathcal{C}_M \wedge \mathcal{C}_B(\mathbf{F}_M) \wedge (\neg(\mathcal{C}(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i})))))$.

[2] We do not need to include the attributes in $\alpha(\mathcal{C}(\mathbf{F}_M))$ in the set as they are all contained in $\alpha(\mathcal{C}) \cup \alpha(\mathcal{C}_B(\mathbf{F}_M))$.

Before we state the corresponding theorem we give a lemma that will simplify the proof of the theorem.

Lemma 4.2 Consider an update operation MODIFY$(R_u, C_M, \mathbf{F}_M)$ and a derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$ where $R_u \in \mathbf{R}$. Given two tuples, t_1 and t_2, over $\alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))$ then there are only four assignments of truth values to the expressions in

$$(\neg(C(\mathbf{F}_M)[t_1] \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})} (A_i = \rho(f_{A_i}))[t_1])))$$
$$\wedge (\neg(C(\mathbf{F}_M)[t_2] \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})} (A_i = \rho(f_{A_i}))[t_2])))$$
$$\wedge (\neg(C(\mathbf{F}_M)[t_1] \Leftrightarrow C(\mathbf{F}_M)[t_2])) \quad (2)$$

which will make this statement evaluate to *true*.

Proof: There are four component expressions in Expression 2, each representing a column of Table 3. Since each may take a truth value from the set {*true*, *false*} this yields sixteen possible assignments. Since $(\neg(C(\mathbf{F}_M)[t_1] \Leftrightarrow C(\mathbf{F}_M)[t_2]))$ must be *true* then the values given to $C(\mathbf{F}_M)[t_1]$ and $C(\mathbf{F}_M)[t_2]$ must be different. This eliminates eight of the sixteen assignments. Also, both factors of the conjunction in $(\neg(C(\mathbf{F}_M)[t_1] \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))[t_1])))$ cannot be *true* if this condition is to be satisfied; this eliminates two more of the remaining eight assignments. Similarly, the condition $(\neg(C(\mathbf{F}_M)[t_2] \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))[t_2])))$ eliminates a further two assignments. Hence, the only assignments that will satisfy Expression 2 are the four given in Table 3. □

$C(\mathbf{F}_M)[t_1]$	$(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i})))[t_1]$	$C(\mathbf{F}_M)[t_2]$	$(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i})))[t_2]$
true	*false*	*false*	*false*
true	*false*	*false*	*true*
false	*false*	*true*	*false*
false	*true*	*true*	*false*

Table 3: Assignments which satisfy Expression 2

Theorem 4.5 *If the operation* MODIFY$(R_u, C_M, \mathbf{F}_M)$ *is autonomously computable with respect to the derived relation defined by* $E = (\mathbf{A}, \mathbf{R}, C)$*, where* E *and the update operation satisfy Properties 1 and 3, then Property 4 must be satisfied.*

Proof: Assume that the update is autonomously computable, that Property 3 is satisfied but that Property 4 is not. This means that $[\alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$ contains an attribute x that is computationally essential in $C(\mathbf{F}_M)$ with respect to the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge [\neg(C(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))]$. We construct two tuples t_1 and t_2 over the attributes

in $\mathbf{A}^+ \cup \alpha(\mathcal{C}) \cup \alpha(\mathcal{C}_M) \cup \alpha(\mathcal{C}_B(\mathbf{F}_M))$. This is done in such a way that $t_1[x] \neq t_2[x]$ but they agree on the values of all other attributes. We require both t_1 and t_2 to satisfy the conditions $\mathcal{C}, \mathcal{C}_M, \mathcal{C}_B(\mathbf{F}_M)$, and $[\neg(\mathcal{C}(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))]$. Since we wish x to be computationally essential we would also like $(\neg(\mathcal{C}(\mathbf{F}_M)[t_1] \Leftrightarrow \mathcal{C}(\mathbf{F}_M)[t_2]))$ to be satisfied. Therefore, by Lemma 4.2 there are four possible assignments to consider. Due to the symmetry of the truth values in these four assignments (see Table 3) we can, without loss of generality, require that t_1 satisfy $\mathcal{C}(\mathbf{F}_M)$, and t_2 not satisfy $\mathcal{C}(\mathbf{F}_M)$. Thus $(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i})))[t_2]$ may be either *true* or *false*; it will not affect the rest of the proof. The fact that x is computationally essential in $\mathcal{C}(\mathbf{F}_M)$ with respect to the condition $\mathcal{C} \wedge \mathcal{C}_M \wedge \mathcal{C}_B(\mathbf{F}_M) \wedge (\neg(\mathcal{C}(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i})))))$ guarantees that such values exist. We can now extend t_1 and t_2 to obtain two different database instances where each relation contains only one tuple. In both cases the derived relation contains the same tuple and the tuple is selected for modification. In one case (for the instance obtained from t_2) the single tuple in the derived relation should be deleted after the modification, while in the other case it should not. Hence, by Lemma 4.1, the update $\mathcal{U}$ is not autonomously computable. □

The only tuples still of interest are those that satisfy the conditions characterized by line four of Table 2 and are in the modify set. These tuples are visibly modified and remain in the updated instance so we need to be able to determine the updated values of all visible attributes. The next property and theorem establish the necessary conditions under which the modified values for the attributes in $\mathbf{A}$ can be correctly computed. But, before we state them we need a new definition which extends the concept of uniquely determined to apply to functions.

Definition 4.6 Consider a set of variables $\{x_1, \ldots, x_j, x_{j+1}, \ldots, x_k\}$. Let $\mathcal{C}$ be a Boolean expression over some of the variables in this set, that is, $\alpha(\mathcal{C}) \subseteq \{x_1, \ldots, x_k\}$. As well, let f represent a function over variables in the set $\{x_1, \ldots, x_k\}$. The value of the function f is said to be *uniquely determined* by condition $\mathcal{C}$ and the set of variables $\{x_1, \ldots, x_j\}$ if

$$\begin{aligned}
&\forall\, x_1, \ldots, x_j, x_{j+1}, \ldots, x_k, x'_{j+1}, \ldots, x'_k \\
&\quad [\mathcal{C}(x_1, \ldots, x_j, x_{j+1}, \ldots, x_k) \wedge \mathcal{C}(x_1, \ldots, x_j, x'_{j+1}, \ldots, x'_k) \\
&\quad \Rightarrow\ (f(x_1, \ldots, x_j, x_{j+1}, \ldots, x_k) = f(x_1, \ldots, x_j, x'_{j+1}, \ldots, x'_k))].
\end{aligned}$$

□

Example 4.6 Let $\mathcal{C} \equiv (H = 5)(I = 10 - J)$ and $f(I, J) = (I + J)$. For any values of I and J that satisfy $\mathcal{C}$ we are guaranteed that the value of $I + J$, and hence f, is 10. In other words the condition of Definition 4.6 becomes

$$\begin{aligned}
&\forall\ \ H, I, J, I', J' \\
&\quad [(H = 5)(I = 10 - J) \wedge (H = 5)(I' = 10 - J') \\
&\quad \Rightarrow\ ((I + J) = (I' + J'))].
\end{aligned}$$

As this is a valid implication we conclude that f is uniquely determined by $\mathcal{C}$. Note, that we can state this in spite of the fact that we do not know the value of either I or J. □

Property 5 Given the update operation MODIFY$(R_u, C_M, \mathbf{F}_M)$ and the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, the value of the function $\rho(f_{A_i})$ is uniquely determined by the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge C(\mathbf{F}_M) \wedge (\neg(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))$ and the attributes in $\mathbf{A}^+$, for each $A_i \in (\alpha(R_u) \cap \mathbf{A})$.

Theorem 4.6 *If the operation* MODIFY$(R_u, C_M, \mathbf{F}_M)$ *is autonomously computable with respect to the derived relation defined by* $E = (\mathbf{A}, \mathbf{R}, C)$*, where* E *and the update operation satisfy Properties 1, 3, and 4, then Property 5 must be satisfied.*

Proof: Assume that $\mathcal{U}$ is autonomously computable, that $\mathbf{A} \cap \alpha(R_u)$ contains a single attribute A_j with a non-trivial f_{A_j}, and that $\rho(f_{A_j})$ is not uniquely determined by the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge C(\mathbf{F}_M) \wedge (\neg(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))$ and the attributes in $\mathbf{A}^+$. We can then construct two tuples t_1 and t_2 over the attributes in $\alpha(R_u) \cup \alpha(C)$ such that t_1 and t_2 both satisfy $C, C_M, C_B(\mathbf{F}_M), C(\mathbf{F}_M)$, and $(\neg(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))$. We also require that t_1 and t_2 agree on the values of all attributes in $\mathbf{A}$; but have some values that make $(\rho(f_{A_j}))[t_1] \neq (\rho(f_{A_j}))[t_2]$. That such a set of values exists is guaranteed by the fact that Property 5 is not satisfied. To find such values one can use the set of values returned by an appropriate satisfiability algorithm used to test the validity of the implication in Definition 4.6. Since $\rho(f_{A_j})$ is not uniquely determined then the implication is *false* when tested for this function. Therefore, the set of values returned by the algorithm will satisfy the antecedent but not the consequent. This is exactly what is required of the values in the tuples t_1 and t_2.

Each of t_1 and t_2 can now be extended into an instance of D, where each relation contains a single tuple. Both instances will give the same instance of the derived relation, consisting of a single tuple $t_1[\mathbf{A}] = t_2[\mathbf{A}]$. In both instances the tuple in the derived relation should be modified; as both tuples satisfy $(\neg(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))$ the modifications will be visible in both cases. However, the value of the modified attribute, A_j, will be different depending on whether we use t_1 or t_2. Hence, by Lemma 4.1, $\mathcal{U}$ is not autonomously computable. □

4.4.3 Updating the Instance

In the previous two sections we gave a number of necessary conditions for an update to be autonomously computable. We will now prove that taken together those conditions are sufficient to guarantee that the update is autonomously computable. However, we first present some notation and then a lemma that will aid in the part of the proof of sufficiency which deals with Property 3.

Recall that the first part of the condition in Property 3 is of the form

$$(C \wedge C_M \wedge C_B(\mathbf{F}_M)) \Rightarrow (C(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})} (A_i = \rho(f_{A_i})))).$$

Let C_{ms} represent the negation of this expression, that is,

$$C_{ms} \equiv C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge (\neg(C(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})} (A_i = \rho(f_{A_i})))))).$$

We will use C_{ms} to determine whether or not a tuple should be placed in the modify set; hence the subscript *ms*. Recall that only those tuples which may be deleted or *visibly* changed by the MODIFY operation constitute the modify set.

Lemma 4.3 Consider a modify operation $\mathcal{U} = \text{MODIFY}(R_u, C_M, \mathbf{F}_M)$, and a derived relation $E = (\mathbf{A}, \mathbf{R}, C)$ where $\mathcal{U}$ and E satisfy Properties 1 and 3. Let t be a tuple in the current instance of E. Tuple t should be placed in the modify set for $\mathcal{U}$ if and only if $C_{ms}[t]$ is satisfiable.

Proof: (Sufficiency) Assume that $C_{ms}[t]$ is satisfiable. For this to be the case each of the expressions used to form C_{ms} must be satisfiable. Hence, there exists a tuple s over $\mathbf{R}$ such that $s[\mathbf{A}] = t$ and s satisfies each expression in C_{ms}. Since $C[s] \wedge C_M[s] \wedge C_B(\mathbf{F}_M)[s]$ is satisfied then t is in the current instance of E and is chosen for modification. In addition, since $(\neg(C(\mathbf{F}_M)[s] \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))[s])))$ is satisfied then at least one of $C(\mathbf{F}_M)[s]$ or $(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))[s])$ is false. Hence, either t will be deleted from the instance after it is modified or it will appear in the new instance but with visible modifications. In either case, t should be placed in the modify set.

(Necessity) Assume that t should be in the modify set but that $C_{ms}[t]$ is not satisfiable. Hence, for every tuple s over $\mathbf{R}$ such that $s[\mathbf{A}] = t$ we have $C_{ms}[s] = \mathit{false}$. This means that at least one of the conjuncts will be *false* when evaluated for tuple s. If $C[s]$ is *false* then t is not in the current instance of E and hence, should not be in the modify set. If $C_M[s] \wedge C_B(\mathbf{F}_M)[s]$ is *false* then t is not chosen for modification; again, t should not be in the modify set. Finally, if $(\neg(C(\mathbf{F}_M)[s] \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))[s])))$ is *false* then both $C(\mathbf{F}_M)[s]$ and $(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))[s])$ must be *true*. In this case t is guaranteed to be in the new instance of E and also to not have any visible modifications. Therefore, again there is no need for t to be in the modify set. Each of the three possibilities reaches a contradiction, therefore we conclude that our assumption is incorrect. Hence, the satisfiability of $C_{ms}[t]$ is a necessary condition. □

Theorem 4.7 *Consider a modify operation* $\mathcal{U} = \text{MODIFY}(R_u, C_M, \mathbf{F}_M)$ *and a derived relation* $E = (\mathbf{A}, \mathbf{R}, C)$ *where* $\mathcal{U}$ *and* E *satisfy Property 1. If Properties 2, 3, 4, and 5 are satisfied then the effect of* $\mathcal{U}$ *is autonomously computable on* E.

Proof: We prove the sufficiency of Properties 2, 3, 4, and 5 by demonstrating how $v(E, d')$ can be determined from $\mathcal{U}$ and $v(E, d)$.

Assume that Property 2 is satisfied. Consider a tuple t in the Cartesian product of the relations in the base $\mathbf{R}$, and assume that t is selected for modification. Let t' denote the corresponding tuple after modification. Assume that t does not satisfy C and hence will not have created any tuple in the derived relation. Because Property 2 is satisfied for every tuple, it must also hold for t and hence t' cannot satisfy C. Consequently, modifying t to t' does not cause any new tuple to appear in the derived relation. Therefore, the only tuples in $v(E, d')$ are those whose unmodified versions are in $v(E, d)$. In other words, we are assured that $v(E, d)$ contains all the tuples needed to compute $v(E, d')$.

Now consider Property 3. If t is a tuple in $v(E, d)$ then, by Lemma 4.3, testing $C_{ms}[t]$ will determine whether or not t should be placed in the modify set. Since this is true for each t in $v(E, d)$, we can decide, for each tuple whether or not it should be in the modify set.

For Property 4, if every attribute $x \in [\alpha(\mathcal{C}) \cup \alpha(\mathcal{C}_M) \cup \alpha(\mathcal{C}_B(\mathbf{F}_M))] - \mathbf{A}^+$ is computationally nonessential in $\mathcal{C}(\mathbf{F}_M)$ with respect to $\mathcal{C} \wedge \mathcal{C}_M \wedge \mathcal{C}_B(\mathbf{F}_M) \wedge (\neg(\mathcal{C}(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i})))))$ we can correctly evaluate the condition $\mathcal{C}(\mathbf{F}_M)$ on every tuple of the modify set by assigning surrogate values to the attributes in $\alpha(\mathcal{C}(\mathbf{F}_M)) - \mathbf{A}^+$. This means that, for those tuples in the modify set we can determine which tuples will remain in the new instance of $v(E, d)$ and which must be deleted.

Finally, assume that Property 5 is satisfied. Hence, for each t, in $v(E, d)$ which is visibly modified and will remain in the new instance, and for each $A_i \in \mathbf{A} \cap \alpha(R_u)$, the value of the expression $\rho(f_{A_i})$ is uniquely determined by the condition $\mathcal{C} \wedge \mathcal{C}_M \wedge \mathcal{C}_B(\mathbf{F}_M) \wedge \mathcal{C}(\mathbf{F}_M) \wedge (\neg(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))$ and the attributes in $\mathbf{A}^+$. A procedure for computing uniquely determined values can be found in [12] and in more detail in [18]. As $\rho(f_{A_i})$ gives the new value for attribute A_i, this means that the given condition and the visible attributes contain sufficient information to determine the updated values of A_i. We have assumed this for each $A_i \in \mathbf{A} \cap \alpha(R_u)$, therefore, the value of every modified attribute in $\mathbf{A}$ is autonomously computable. Hence, for every tuple in $v(E, d)$ which is visibly modified and remains in the updated instance we can calculate the new values of all visible attributes.

The entries in Tables 1 and 2 completely characterize all the possible cases for a tuple t. To distinguish between these cases we have defined four properties. We have shown that these four properties are sufficient to allow us to compute the updated instance from $v(E, d)$, the definition E, and $\mathcal{U}$. The procedure described above defines the function $F_{\mathcal{U},E}$. Hence, the MODIFY is autonomously computable on E. □

We give an example which proceeds through the four steps associated with Theorems 4.3 through 4.7, at each step testing the appropriate condition.

Example 4.7 Suppose a database consists of the two relation schemes $R_1(H, I)$ and $R_2(J, K, L)$ where H, I, J, K and L each have the domain $[0, 30]$. Let the derived relation and modify operation be defined as:

$$E = (\{I, J\}, \{R_1, R_2\}, (H < 15)(I = K)(L = 20))$$
$$\text{MODIFY}(R_2, (K > 5)(K \leq 22), \{(J := L + 3), (K := K), (L := L)\})$$

and note that Property 1 is satisfied since the value of J, which is visible in E, may be modified. From the definition of the derived relation we can see that $\mathbf{A} = \{I, J\}$ and $\mathbf{A}^+ = \{I, J, K, L\}$.

Test of Property 2:

$$\begin{aligned} \forall \;\; & H, I, K, L \; [(\neg((H < 15)(I = K)(L = 20))) \wedge (K > 5)(K \leq 22) \\ & \wedge \; (L + 3 \geq 0)(L + 3 \leq 30)(K \geq 0)(K \leq 30)(L \geq 0)(L \leq 30) \\ & \quad \Rightarrow \; (\neg((H < 15)(I = K)(L = 20)))] \end{aligned}$$

Note that, the consequent $\neg((H < 15)(I = K)(L = 20))$ appears as a condition in the antecedent as well. Therefore, the given implication is valid, and we can conclude that the modify operation will not introduce new tuples into $v(E, d)$.

Test of Property 3:

$$\begin{aligned}
\forall\ & \langle H,I,J,K,L\rangle, \langle H',I,J,K,L\rangle \\
& [[(H<15)(I=K)(L=20) \wedge (K>5)(K\leq 22) \\
& \wedge (L+3\geq 0)(L+3\leq 30)(K\geq 0)(K\leq 30)(L\geq 0)(L\leq 30) \\
& \quad \Rightarrow (\neg((H<15)(I=K)(L=20)\wedge(J=L+3)))] \\
& \vee \\
& [(H<15)(I=K)(L=20)\wedge(H'<15)(I=K)(L=20) \\
& \Rightarrow ((K>5)(K\leq 22)\ \wedge\ (L+3\geq 0)(L+3\leq 30)(K\geq 0)(K\leq 30)(L\geq 0)(L\leq 30) \\
& \quad \Leftrightarrow (K>5)(K\leq 22)\ \wedge\ (L+3\geq 0)(L+3\leq 30)(K\geq 0)(K\leq 30)(L\geq 0)(L\leq 30))]]
\end{aligned}$$

Consider the implication which is after the disjunction; comprising the last three lines. The right-hand-side of this implication contains an equivalence. Since the two conditions in the equivalence are identical, then clearly, the implication is valid. Thus we can correctly select the tuples in the derived relation that satisfy $C_M \wedge C_B(\mathbf{F}_M)$ and will be visibly modified.

Test of Property 4:

We are now interested in the set, $[\alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+ = \{H,I,K,L\} - \{I,J,K,L\} = \{H\}$.

$$\begin{aligned}
\forall\ & H,I,J,K,L,H'\ [(H<15)(I=K)(L=20)\wedge(K>5)(K\leq 22) \\
& \wedge\ (L+3\geq 0)(L+3\leq 30)(K\geq 0)(K\leq 30)(L\geq 0)(L\leq 30) \\
& \wedge\ (\neg(((H<15)(I=K)(L=20)\wedge(J=L+3))) \\
& \wedge\ (H'<15)(I=K)(L=20)\wedge(K>5)(K\leq 22) \\
& \wedge\ (L+3\geq 0)(L+3\leq 30)(K\geq 0)(K\leq 30)(L\geq 0)(L\leq 30) \\
& \wedge\ (\neg(((H'<15)(I=K)(L=20)\wedge(J=L+3))) \\
& \quad \Rightarrow ((H<15)(I=K)(L=20) \Leftrightarrow (H'<15)(I=K)(L=20))]
\end{aligned}$$

The conditions $(H<15)$ and $(H'<15)$ in the antecedent guarantee that the expressions in the consequent will be equivalent. Therefore, H is computationally nonessential in $C(\mathbf{F}_M)$ with respect to $C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge (\neg(C(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u)\cap \mathbf{A})}(A_i = \rho(f_{A_i})))))$. Thus, determining the tuples, in the modify set of the current instance, which satisfy $C(\mathbf{F}_M)$ can be computed autonomously.

Test of Property 5:

$$\begin{aligned}
\forall\ & H,I,K,L,H'\ [(H<15)(I=K)(L=20)\wedge(K>5)(K\leq 22) \\
& \wedge\ (L+3\geq 0)(L+3\leq 30)(K\geq 0)(K\leq 30)(L\geq 0)(L\leq 30) \\
& \wedge\ (H<15)(I=K)(L=20)\wedge(\neg(J=L+3)) \\
& \wedge\ (H'<15)(I=K)(L=20)\wedge(K>5)(K\leq 22) \\
& \wedge\ (L+3\geq 0)(L+3\leq 30)(K\geq 0)(K\leq 30)(L\geq 0)(L\leq 30) \\
& \wedge\ (H'<15)(I=K)(L=20)\wedge(\neg(J=L+3)) \\
& \quad \Rightarrow ((L+3)=(L+3))]
\end{aligned}$$

By considering the right-hand-side of this implication we see that it is obviously valid. Hence, the above condition verifies that the expression f_J is uniquely determined by the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge C(\mathbf{F}_M) \wedge (\neg(\bigwedge_{A_i \in (\alpha(R_u)\cap \mathbf{A})}(A_i = \rho(f_{A_i}))))$ and the variables $\mathbf{A}^+$. Therefore, the

new attribute values for the visibly modified tuples that will remain in $v(E,d)$ are autonomously computable.

To summarize, let us consider a numeric example for the given database scheme.

Before

r_1: H	I
1	5
2	15
3	22
4	20

r_2: J	K	L
19	5	20
10	15	29
16	22	20
18	20	29

$v(E,d)$: I	J
5	19
22	16

After

r_1: H	I
1	5
2	15
3	22
4	20

r_2: J	K	L
19	5	20
32	15	29
23	22	20
32	20	29

$v(E,d')$: I	J
5	19
22	23

Property 2 provides assurance that the tuples in the Cartesian product of r_1 and r_2, which do not satisfy C before modification, will not satisfy C after. For example, consider the last tuple in each of r_1 and r_2. In the Cartesian product these will form $\langle 4, 20, 18, 20, 29\rangle$. Although this does satisfy $(I = K)$ it does not satisfy $(L = 20)$ and hence will not be in the current instance. Moreover, even though this tuple is modified to become $\langle 4, 20, 32, 20, 29\rangle$ it still does not satisfy $(L = 20)$ and, hence, remains outside the updated derived relation instance.

Property 3 guarantees that we can determine which tuples in $v(E,d)$ belong in the modify set. For each tuple in $v(E,d)$ we need to test the satisfiability of

$$C_{ms} \equiv C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge (\neg(C(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})} (A_i = \rho(f_{A_i}))))).$$

For this example we have

$$\begin{aligned} C_{ms} \equiv\ & [(H < 15)(I = K)(L = 20) \wedge (K > 5)(K \leq 22) \\ & \wedge (L + 3 \geq 0)(L + 3 \leq 30)(K \geq 0)(K \leq 30)(L \geq 0)(L \leq 30) \\ & \wedge (\neg((H < 15)(I = K)(L = 20) \wedge (J = L + 3)))]. \end{aligned}$$

The first tuple in $v(E,d)$ gives us $\langle h, 5, 19, k, l\rangle$ to test. Since, $C_{ms}[\langle h, 5, 19, k, l\rangle]$ is not satisfiable we conclude that $\langle 5, 19\rangle$ will not be visibly modified and can be placed in the new instance. The second tuple to test from $v(E,d)$ is $\langle h, 22, 16, k, l\rangle$. Since, $C_{ms}[\langle h, 22, 16, k, l\rangle]$ is satisfiable we conclude that $\langle 22, 16\rangle$ belongs in the modify set.

Property 4 allows us to determine which tuples in the modify set will be deleted since they will no longer satisfy condition C. The tuple $\langle 22, 16\rangle$ satisfies $C(\mathbf{F}_M)$ and will remain in the updated instance[3].

[3] Since the attributes in C have trivial update functions $C(\mathbf{F}_M) \equiv C$. Hence, no tuple will be deleted from the derived relation.

Property 5 ensures that we can compute the new values for the modified tuples. Here we need to compute $(L + 3)$. Since C contains the condition $(L = 20)$ we know that for any tuple in the modify set $(L + 3) = 23$. Therefore, $\langle 22, 16\rangle$ should be updated to $\langle 22, 23\rangle$ in the new instance. □

5 Implementation Issues

As pointed out earlier we assume the existence of an algorithm to test the satisfiability of a particular class of Boolean expressions. We may not be able, in general, to use that algorithm to test the condition of Definition 4.6. This is due to the fact that when we substitute an update function in the right-hand-side of the implication given in that definition we may obtain atomic conditions that the algorithm cannot handle. However, the following corollary establishes a sufficient condition, that allows us to compute the new values of the attributes in $\mathbf{A}$, for which we can still use that algorithm.

Recall that $\alpha(\rho(f_{A_i}))$ denotes the set of attributes occurring in the right hand side expression of f_{A_i}. Define the set Z as

$$Z = \bigcup_{A_i \in \mathbf{A} \cap \alpha(R_u)} \alpha(\rho(f_{A_i}))$$

that is, Z is the set of attributes used to compute the new values for the attributes in $\mathbf{A}$.

Corollary 5.1 *If all variables in* $Z - \mathbf{A}^+$ *are uniquely determined by the condition* $C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge C(\mathbf{F}_M) \wedge (\neg(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))$ *and the attributes in* $\mathbf{A}^+$ *then Property 5 is satisfied.*

Proof: Assume the values of the attributes in $Z - \mathbf{A}^+$ are uniquely determined by the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge C(\mathbf{F}_M) \wedge (\neg(\bigwedge_{A_i \in (\alpha(R_u) \cap \mathbf{A})}(A_i = \rho(f_{A_i}))))$ and the attributes in $\mathbf{A}^+$. This means that we can determine (exactly) one value for each argument of each update function. Since each $\rho(f_{A_j})$ is a *function* it will have a single value for the set of arguments so determined. □

In practice it is probably better to enforce the conditions of Corollary 5.1 instead of those of Property 5.

Consider the condition stated in Property 3. We consider that occurrences of tuples for which the first disjunct of the condition is *true* but the second disjunct is *false* will be very rare. Put more fully, assume this situation occurs for some tuple t in the derived relation instance. Then we know that if t is modified we can *prove* that it will remain in the instance and that it will *not* be visibly changed; moreover, we will *not* be able to evaluate $C_M[t] \wedge C_B(\mathbf{F}_M)[t]$. Although such situations must be considered in a theoretical discussion it seems unprofitable to test for them in practical situations. Therefore, we present the following property, which states a condition which is sufficient but not necessary.

Property 3′ Given the update operation MODIFY$(R_u, C_M, \mathbf{F}_M)$ and the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, the attributes in $[\alpha(C_M) \cup \alpha(C) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$ are computationally nonessential in $C_M \wedge C_B(\mathbf{F}_M)$ with respect to C.

Of course, if we use this instead of Property 3 we cannot use Properties 4 and 5 without adjusting their conditions. We give the corresponding properties.

Property 4' Given the update operation MODIFY$(R_u, C_M, \mathbf{F}_M)$ and the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, the attributes in the set $[\alpha(C) \cup \alpha(C_M) \cup \alpha(C_B(\mathbf{F}_M))] - \mathbf{A}^+$ are computationally nonessential in $C(\mathbf{F}_M)$ with respect to the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M)$.

Property 5' Given the update operation MODIFY$(R_u, C_M, \mathbf{F}_M)$ and the derived relation defined by $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, for each $A_i \in \mathbf{A} \cap \alpha(R_u)$, the value of the function $\rho(f_{A_i})$ is uniquely determined by the condition $C \wedge C_M \wedge C_B(\mathbf{F}_M) \wedge C(\mathbf{F}_M)$ and the attributes in $\mathbf{A}^+$.

We state the following theorem without proof.

Theorem 5.2 *Consider a modify operation* $\mathcal{U}$ = MODIFY$(R_u, C_M, \mathbf{F}_M)$ *and a derived relation* $E = (\mathbf{A}, \mathbf{R}, C)$ *where* $\mathcal{U}$ *and* E *satisfy Property 1. If Properties 2, 3', 4', and 5' are satisfied then the effect of* $\mathcal{U}$ *is autonomously computable on* E. □

Of course, just as Corollary 5.1 provides a sufficient condition for Property 5 to be satisfied, we can provide a sufficient condition which guarantees Property 5' will be satisfied. We state it without proof.

Corollary 5.3 *If all variables in* $Z - \mathbf{A}^+$ *are uniquely determined by the condition* $C \wedge C_M \wedge C(\mathbf{F}_M) \wedge C_B(\mathbf{F}_M)$ *and the attributes in* $\mathbf{A}^+$ *then Property 5' is satisfied.* □

Given a MODIFY operation and a derived relation defined by an expression E, we first test whether or not the update is irrelevant and then (if it is not irrelevant) we proceed to test whether or not the update is autonomously computable. The test for irrelevant updates is stated by Theorem 3.3 and the first test for autonomously computable modifications is given by Theorem 4.3. These two tests can be cascaded as explained below.

The condition for irrelevant modifications is given by

$$\forall[C_M \wedge C_B(\mathbf{F}_M) \Rightarrow (\neg C \wedge \neg C(\mathbf{F}_M)) \vee (C \wedge C(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in \mathcal{I}} (A_i = \rho(f_{A_i}))))]$$

where $\mathcal{I} = \mathbf{A} \cap \alpha(R_u)$. This is equivalent to testing

$$\forall[(\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow \neg C(\mathbf{F}_M)) \wedge \left(C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow C(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in \mathcal{I}} (A_i = \rho(f_{A_i})))\right)]$$

which is equivalent to testing

$$cl\forall[\neg C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow \neg C(\mathbf{F}_M)] \wedge \tag{3}$$

$$\forall[C \wedge C_M \wedge C_B(\mathbf{F}_M) \Rightarrow C(\mathbf{F}_M) \wedge (\bigwedge_{A_i \in \mathcal{I}} (A_i = \rho(f_{A_i})))]. \tag{4}$$

Notice that the first condition of the outermost "∧" operator is exactly the condition stated by Property 2 and the second condition is the first implication in Property 3. Therefore, we can test for irrelevant modifications by testing, in two stages, the above condition and storing the two results. To test for autonomously computable modifications we can avoid testing the condition of Property 2 by recalling the first result from the test for irrelevant modifications. Similarly, if the result of the second test is *true* we know that Properties 3, 4, and 5 must be satisfied. The last two properties will be satisfied vacuously since the modify set is empty. However, if the result of the second test is *false* then we must test the entire condition of Property 3.

6 Experimental Results

In order to test the viability of the approach presented we have built a prototype system. The prototype accepts as input a database schema, definitions of derived relations, and update operations. For each operation given it tests the update against each of the derived relations. The outcome of this test is a partitioning of the set of derived relations into three classes; those derived relations for which the update operation is irrelevant, those for which it is autonomously computable, and those for which it is neither irrelevant nor autonomously computable. The tests for autonomously computable modifications is based on Properties 2, 3′, 4′, and Corollary 5.3. This section reports the cost of testing a number of update operations using the prototype system.

The example database used to obtain the experimental results reported below represents an enterprise which keeps information on customers, distributors, items, and orders for items. Each customer and distributor is located in a particular region. The derived relations used suggest a distributed database where the regions are grouped into three geographic areas. This is reflected in the derived relations by partitioning the relations based on region. We wished to allow some variety in the type of derived relation used. To this end we did not horizontally partition the join of orders and items to obtain three derived relations, but left this as a single derived relation. We could view this as a ramification of the fact that order processing is centralized. As well, one derived relation corresponds exactly to one of the conceptual relations. There are a total of six conceptual relations and seventeen derived relations. The details of the conceptual and derived relations as well as the update operations are given in Appendix B.

The times where taken with the prototype running on a VAX 8650[4] under the UNIX[5] operating system. The code is written in the C programming language. Although an attempt was made to avoid obvious inefficiencies, none was made to produce "optimized" code. The point of the prototyping exercise was to demonstrate the feasibility of the approach, not to produce code of commercial quality.

Some explanation of the contents of table 4 is in order. The update numbers refer to the update operations listed in Appendix B. The three columns entitled "Number of Derived Relations" give

[4]VAX is a Trademark of Digital Equipment Corporation.

[5]UNIX is a Trademark of AT&T Bell Laboratories.

Update Number	Update Type	Number of Derived Relations			Time required (ms)		
		Irrelevant	Aut.Comp.	Neither	Irrelevant	Aut.Comp.	Total
U1	INSERT	11(11)	0	6	53	0	53
U2	MODIFY	11(11)	3	3	405	54	459
U3	DELETE	7(7)	10	0	75	9	84
U4	DELETE	7(7)	10	0	78	9	87
U5	DELETE	7(7)	10	0	106	9	115
U6	MODIFY	14(8)	3	0	239	34	273
U7	MODIFY	11(8)	3	3	197	34	231
U8	MODIFY	14(8)	3	0	325	37	362
U9	MODIFY	14(8)	3	0	247	36	283
U10	DELETE	10(10)	7	0	59	7	66
U11	MODIFY	12(12)	5	0	309	65	374
U12	MODIFY	12(12)	5	0	276	61	337
U13	INSERT	13(13)	0	4	40	0	40
U14	MODIFY	13(13)	4	0	379	77	456

Table 4: Results of testing updates against 17 derived relations.

the number of derived relations in each class as discussed above. The figures in parentheses indicate the number of derived relations for which the update is *trivially* irrelevant. Similarly, the columns entitled "Time required" give the number of milliseconds (ms) required to determine for which derived relations the update is irrelevant, for which of the remaining derived relations the update is autonomously computable, and the total time to perform these tests. The time required to determine trivially irrelevant updates has not been listed as it is usually less than 1ms.

For example, consider the results given for update U7; it modifies the region of some distributors. It is irrelevant to eleven derived relations; the bracketed value indicates that it is trivially irrelevant to eight of these. Of the remaining six derived relations U7 is autonomously computable on three. This means that there are three derived relations for which U7 is neither irrelevant nor autonomously computable. Testing to find out which of the seventeen derived relations U7 is irrelevant to took 197ms. Testing the remaining six derived relations to determine on which U7 is autonomously computable took 34ms. Hence the total time for testing was 231ms.

The modify operations required the most time to test; this is not surprising considering the number and complexity of the conditions to be tested. Even so, in the worst case the total time required for a modification was less than 0.5s and for all the updates tested the average was about 0.23s. We believe that these examples demonstrate the practicality of our approach and justify continued research in this area.

The times listed in Table 4 only reflect the time required to determine which class each derived

relation is in for this update operation. Additional computation, involving the retrieval and updating of tuples in the database would be required to actually perform the update operation, even when it is autonomously computable.

7 Conclusion

Necessary and sufficient conditions for detecting when an update operation is irrelevant to a derived relation (or view, or integrity constraint) have not previously been available for any nontrivial class of updates and derived relations. The concept of autonomously computable updates is completely new. Limiting the class of derived relations to those defined by PSJ-expressions does not seem to be a severe restriction, at least not as it applies to structuring the stored database in a relational system. The update operations considered are fairly general. In particular, this seems to be one of a few papers on update processing where modify operations are considered explicitly and separately from insert and delete operations. Previously, modifications have commonly been treated as a sequence of deletions followed by insertion of the modified tuples.

Testing the conditions given in the theorems above is efficient in the sense that it does not require retrieval of any data from the database. According to our definitions, if an update is irrelevant or autonomously computable, then it is so for *every* instance of the base relations. The fact that an update is not irrelevant does not necessarily mean that it will affect the derived relation. Determining whether or not it will, requires checking the current instance. The same applies for autonomously computable updates.

It should be emphasized that the theorems hold for any class of Boolean expressions. However, actual testing of the conditions requires an algorithm for proving the satisfiability of Boolean expressions. Currently, efficient algorithms exist only for a restricted class of expressions, the restriction being on the atomic conditions allowed. An important open problem is to find efficient algorithms for more general types of atomic conditions. The core of such an algorithm is a procedure for testing whether a set of inequalities/equalities can all be simultaneously satisfied. The complexity of such a procedure depends on the type of expressions (functions) allowed and the domains of the variables. If linear functions with variables ranging over the real numbers (integers) are allowed, the problem is equivalent to finding a feasible solution to a linear programming (integer programming) problem.

We have not imposed any restrictions on valid instances of base relations, for example, functional dependencies or inclusion dependencies. Any combination of attribute values drawn from their respective domains represents a valid tuple. Any set of valid tuples is a valid instance of a base relation. If relation instances are further restricted, then the given conditions are still sufficient, but they may not be necessary.

In particular, how to make use of referential integrity constraints is an important open problem. At present the system has no knowledge of such constraints; this leads to situations where the results of a test could be more informative than is now the case. The following example based on the database in Appendix B illustrates the problem. It would not be unusual to demand that

a customer tuple be present in the Customer relation before an order tuple can appear in the Order relation for the corresponding customer number. Let us consider what would happen in our system if we inserted a new tuple, <custNumb=456,custName="XYZ Corp.",custRegn=36>, in the Customer relation. As a result of testing, the system would determine that the update is not irrelevant to the derived relations PartOrder, CustWest, OrdrWest, and FillWest; and is irrelevant to all others. However, referential integrity implies that only one of these four derived relations—CustWest—can actually be affected by this update. Trying to determine how this update affects the other three is therefore a waste of time.

Appendix A: Satisfiability Algorithm

The theorems presented in this paper require that statements be proven at *run-time*, that is, when updates are issued. What is required is that certain types of Boolean expressions be tested for unsatisfiability or that implications involving Boolean expressions be proven valid. The latter problem can be translated into one of showing that a Boolean expression is unsatisfiable. Hence, in either case we can proceed by testing satisfiability.

Rosenkrantz and Hunt [17] gave an algorithm for testing the satisfiability of conjunctive Boolean expressions where the atomic conditions come from a restricted class. Their algorithm is based on Floyd's all-pairs-shortest-path algorithm [9] and therefore has an $O(n^3)$ worst case complexity, where n is the number of distinct variables in the expression. The algorithm presented here is a modification of that given by Rosenkrantz and Hunt; there are three main differences. First, we assume that each variable has a finite domain whereas Rosenkrantz and Hunt allow infinite domains. Second, if the expression is satisfiable our algorithm not only verifies the satisfiability but also produces an assignment of values to the variables which satisfies the expression. Third, although the worst case complexity of our algorithm remains $O(n^3)$, for some cases the complexity is reduced to $O(n^2)$.

The algorithm given here tests the satisfiability of a restricted class of Boolean expressions. Each variable is assumed to take its values from a finite, ordered set. Since there is an obvious mapping from such sets to the set of integers, we always assume that the domain consists of a finite interval of the integers. It is assumed that each Boolean expression, B, over the variables $x_1, x_2, \ldots x_n$, is in conjunctive form, i.e. $B = B_1 \wedge B_2 \wedge \ldots \wedge B_m$, and that each atomic condition, B_i, is of the form $(x_i\ \theta\ x_j + c)$ or $(x_i\ \theta\ c)$ where $\theta \in \{=, <, \leq, >, \geq\}$ and c is an integer constant. We will outline the algorithm and discuss each step in more detail.

(Step 1) We first must *normalize* B so that the resulting expression, $N = N_1 \wedge N_2 \wedge \ldots \wedge N_{m_N}$, only has atomic conditions of the form $(x_i \leq x_j + c)$. Conditions of the form $(x_i\ \theta\ c)$ are handled by modifying the domain bounds for the variable x_i. If any such modification results in $U[i] < L[i]$, for any $1 \leq i \leq n$, then B is unsatisfiable.

(Step 2) We build a weighted, directed graph $G = (V, E)$ representing N. Without loss of generality we assume that N is defined over the variables $x_1, \ldots, x_{n_N}$ where $n_N \leq n$. Each variable in N is represented by a node in G. For the atomic condition $(x_i \leq x_j + c)$ we construct an arc from node "x_j" to node "x_i" having weight c. Hence, $|V| = n_N$ and $|E| = m_N$. The graph is represented by an $n_N \times n_N$ array A where, initially, $A(i, j) = c$ if and only if N contains an expression of the form $(x_i \leq x_j + c)$. If two nodes do not have an arc between them the corresponding array entry is labeled with ∞ (i.e. an arbitrarily large positive value). If there is more than one arc between a pair of nodes, then use the one with lowest weight.

(Step 3) The graph, G, can be *reduced* in size by removing nodes of in-degree zero. The justification for doing this is that if x_j is such a node then N does not have any conditions of the form $(x_j \leq x_i + c)$. Hence, the upper limit value of x_j is not constrainted by the value assigned to any other variable. Therefore, we allow x_j to be assigned its (possibly modified) upper bound $U[j]$.

Algorithm **Satisfiable**

Input: A conjunctive Boolean expression B over the variables $x_1, \ldots, x_n$. The variables' domain bounds are given by vectors L and U with $L[i] \leq x_i \leq U[i], 1 \leq i \leq n$.

Output: If B is satisfiable then the value *true* is returned and the assignment $x_i := U[i]$ will satisfy B, otherwise *false* is returned.

Begin

1. Normalize B to obtain N. Check the resulting domain bounds; if $U[i] < L[i]$ for some x_i then return *false.*
2. Initialize A (the adjacency matrix of the directed graph representing N).
3. Reduce A, that is, remove (recursively) rows representing nodes of in-degree zero.
4. Test the trial values against the lower bounds. If $U[i] < L[i]$ for some x_i then return *false.*
5. Test the trial values in N. If they satisfy N return *true.*
6. Execute Floyd's Algorithm on A, after each iteration of the outer loop perform the following:
   (a) If A contains a negative cycle then return *false.*
   (b) Calculate the new trial values.
   (c) Test the trial values against the lower bounds. If $U[i] < L[i]$ for some x_i then return *false.*
   (d) Test the trial values in N. If they satisfy N return *true.*

End

Also, for each node x_i in G, such that there is an arc of weight c from x_j to x_i, we replace the upper bound on x_i with $\min\{U[i], U[j] + c\}$. We can then remove node x_j and its incident arcs from G. This process may be repeated until no nodes of in-degree zero remain in G. If G is acyclic (i.e. if B does not contain a cyclic chain of atomic conditions involving mutually dependent variables) then this process will terminate when there are no edges left in G. The resulting trial assignment of values either proves or disproves that B is satisfiable (Steps 4 and 5). As the reduction process requires examining each element of matrix A at most once, the entire Satisfiable algorithm performs $O(n_N^2)$ operations in this case.

(Step 6) If G is not acyclic then A is used as the input to a modified version of Floyd's algorithm [9] to determine either that N is unsatisfiable or to produce an assignment which satisfies

N. The idea is that we give each remaining variable, x_i, an initial *trial* value equal to its (possibly modified) upper bound $U[i]$. At each iteration we adjust the values (downward) to reflect the current values in A and the previous set of trial values. The iterations continue until we find an assignment to the variables which satisfies N or until we determine that N is unsatisfiable. This takes at most n_N iterations.

To be more specific, given a graph with nodes $x_1, \ldots, x_{n_N}$, the kth step of Floyd's algorithm produces the least weight path between each pair of nodes, with intermediate nodes from the set $\{x_1, x_2, \ldots, x_k\}$. In terms of the Boolean expression this corresponds to forming, from the conditions in N, the most restrictive condition between each pair of variables. The only conditions of N which may be used at the kth step are those involving the variables $x_1, \ldots, x_k$. The new trial value, $U[i]$, for x_i is found by taking $\min\{U[j] + A[i,j]\}$ for $1 \leq j \leq n_N$. There are three possible situations that indicate that the algorithm should terminate. We test each of these conditions after each iteration:

1. Is there a negative weight cycle? In this case N is unsatisfiable.

2. Does the current trial assignment violate any variable's lower bound? Again, N is unsatisfiable.

3. Does the current trial assignment satisfy the lower bound for each variable and satisfy N? In this case N is satisfiable.

Since the longest cycle can contain at most n_N arcs we conclude that this is the maximum number of iterations required; hence the $O(n_N^3)$ complexity. If after n_N iterations we have not found a negative weight cycle or violated any lower bound then the current trial assignment must satisfy N.

Appendix B: Example Database and Updates

This appendix contains the details of the database and updates used to obtain the timings in Section 6. As explained in that section, the enterprise being modelled maintains data about customers, distributors, items, and orders. To that end we use the following six conceptual relations:

Conceptual Relations

Customer (custNumb, custName, custRegn)

Distributor (distNumb, distName, distRegn)

Item (itemNumb, itemDesc, itemPrix)

Order (ordrNumb, ordrDate, ordrCust)

Line (lineOrdr, lineItem, lineQnty)

Available (avlbItem, avlbDist, avlbSply)

The underlined attribute(s) serve as the key of the relation. The first four characters of each attribute name are used to indicate which conceptual relation contains the attribute; the last four are to designate the attribute and perhaps suggest if it is a foreign key. In particular, Numb stands for number, Regn for region, Desc for description, Prix for price, Cust for customer, Ordr for order, Qnty for quantity, Dist for distributor, and Sply for supply. The intention is that lineOrdr and lineItem refer to ordrNumb and itemNumb respectively; avlbItem and avlbDist to itemNumb and distNumb respectively. The first four relations need no comment. The Line relation gives the quantity ordered of a particular item for a particular order. The Available relation gives the supply a particular distributor has of a particular item. Recall that these are the relations which the user "sees" and against which all queries and updates are posed.

The derived relations which are used to actually store the above conceptual relations are designed to efficiently yield information about customers, distributors, orders, availability of items, and which distributors can fill which orders for which items; these are the "Cust", "Dist", "Ordr", "Avlb", and "Fill" derived relations. In keeping with the distributed flavour of this example each of these is actually three derived relations, based on region. The partitioning is done by regions, regions 10 to 19 being East, 20 to 29 Cent(ral), and 30 to 39 West. We envision that the relevant derived relations are stored at three regional offices. It is also assumed that a centralized order processing office is located at the Central office and that the Part and PartOrder derived relations are stored at that office also. Thus, the types of derived relations used are quite varied. The resulting seventeen derived relations are:

Derived Relations

Part = ({itemNumb,itemDesc,itemPrix}, {Item}, (TRUE))

PartOrder = ({itemNumb,itemDesc,itemPrix,lineOrdr,lineItem,lineQnty,
ordrNumb,ordrDate,ordrCust,custNumb,custName,custRegn},
{Item,Line,Order,Customer},
(itemNumb=lineItem)∧(lineOrdr=ordrNumb)∧(ordrCust=custNumb))

CustEast = ({custNumb,custName,custRegn}, {Customer}, (custRegn≥10)∧(custRegn<20))

CustCent = ({custNumb,custName,custRegn}, {Customer}, (custRegn≥20)∧(custRegn<30))

CustWest = ({custNumb,custName,custRegn}, {Customer}, (custRegn≥30)∧(custRegn<40))

DistEast = ({distNumb,distName,distRegn}, {Distributor}, (distRegn≥10)∧(distRegn<20))

DistCent = ({distNumb,distName,distRegn}, {Distributor}, (distRegn≥20)∧(distRegn<30))

DistWest = ({distNumb,distName,distRegn}, {Distributor}, (distRegn≥30)∧(distRegn<40))

OrdrEast = ({ordrNumb,ordrDate,ordrCust,custNumb,custName,custRegn},
{Order,Customer},
(ordrCust=custNumb)∧(custRegn≥10)∧(custRegn<20))

OrdrCent = ({ordrNumb,ordrDate,ordrCust,custNumb,custName,custRegn},
{Order,Customer},
(ordrCust=custNumb)∧(custRegn≥20)∧(custRegn<30))

OrdrWest = ({ordrNumb,ordrDate,ordrCust,custNumb,custName,custRegn},
{Order,Customer},
(ordrCust=custNumb)∧(custRegn≥30)∧(custRegn<40))

AvlbEast = ({itemNumb,itemDesc,itemPrix,avlbItem,avlbDist,avlbSply,distNumb,distName,distRegn},
{Item,Available,Distributor},
(itemNumb=avlbItem)∧(avlbDist=distNumb)∧(distRegn≥10)∧(distRegn<20))

AvlbCent = ({itemNumb,itemDesc,itemPrix,avlbItem,avlbDist,avlbSply,distNumb,distName,distRegn},
{Item,Available,Distributor},
(itemNumb=avlbItem)∧(avlbDist=distNumb)∧(distRegn≥20)∧(distRegn<30))

AvlbWest = ({itemNumb,itemDesc,itemPrix,avlbItem,avlbDist,avlbSply,distNumb,distName,distRegn},
{Item,Available,Distributor},
(itemNumb=avlbItem)∧(avlbDist=distNumb)∧(distRegn≥30)∧(distRegn<40))

FillEast = ({distNumb,distName,distRegn,avlbItem,avlbDist,avlbSply,lineOrdr,lineItem,lineQnty,
ordrNumb,ordrDate,ordrCust,custNumb,custName,custRegn},
{Distributor,Available,Line,Order,Customer},
(distRegn≥10)∧(distRegn<20)∧(distNumb=avlbDist)∧(avlbItem=lineItem)∧(avlbSply≥lineQnty)
∧(lineOrdr=ordrNumb)∧(ordrCust=custNumb)∧(custRegn≥10)∧(custRegn<20))

FillCent = ({distNumb,distName,distRegn,avlbItem,avlbDist,avlbSply,lineOrdr,lineItem,lineQnty,
ordrNumb,ordrDate,ordrCust,custNumb,custName,custRegn},
{Distributor,Available,Line,Order,Customer},
(distRegn≥20)∧(distRegn<30)∧(distNumb=avlbDist)∧(avlbItem=lineItem)∧(avlbSply≥lineQnty)
∧(lineOrdr=ordrNumb)∧(ordrCust=custNumb)∧(custRegn≥20)∧(custRegn<30))

FillWest = ({distNumb,distName,distRegn,avlbItem,avlbDist,avlbSply,lineOrdr,lineItem,lineQnty,
ordrNumb,ordrDate,ordrCust,custNumb,custName,custRegn},
{Distributor,Available,Line,Order,Customer},
(distRegn≥30)∧(distRegn<40)∧(distNumb=avlbDist)∧(avlbItem=lineItem)∧(avlbSply≥lineQnty)
∧(lineOrdr=ordrNumb)∧(ordrCust=custNumb)∧(custRegn≥30)∧(custRegn<40))

The following fourteen updates were used to obtain the timings given in table 4.

Updates

U1 = INSERT(Available, { < avlbItem=43, avlbDist=542, avlbSply=10 >,
< avlbItem=112, avlbDist=571, avlbSply=800 >,
< avlbItem=203, avlbDist=627, avlbSply=250 > })

U2 = MODIFY(Available, (avlbItem=117),
{ avlbItem=avlbItem, avlbDist=avlbDist, avlbSply=avlbSply+300 })

U3 = DELETE(Customer, (custNumb=123))

U4 = DELETE(Customer, (custNumb≥123)∧(custNumb≤130))

U5 = DELETE(Customer,
(custNumb=123)∨(custNumb=125)∨(custNumb=127)∨(custNumb=129))

U6 = MODIFY(Distributor, (distRegn=19),
{ distNumb=distNumb, distName=distName, distRegn=18 })

U7 = MODIFY(Distributor, (distRegn=19),
{ distNumb=distNumb, distName=distName, distRegn=20 })

U8 = MODIFY(Distributor, (distRegn=18)∨(distRegn=19),
{ distNumb=distNumb, distName=distName, distRegn=17 })

U9 = MODIFY(Distributor, (distRegn≥18)∧(distRegn≤19),
{ distNumb=distNumb, distName=distName, distRegn=17 })

U10 = DELETE(Order, (ordrDate<860901))

U11 = MODIFY(Item, (itemNumb=45)∨(itemNumb=113)∨(itemNumb=297),
{ itemNumb=itemNumb, itemDesc=itemDesc, itemPrix=itemPrix+125 })

U12 = MODIFY(Item, (itemNumb=45)∨(itemNumb=113)∨(itemNumb=297),
{ itemNumb=itemNumb, itemDesc=itemDesc, itemPrix=995 })

U13 = INSERT(Line, { < lineOrdr=101, lineItem=42, lineQnty=3 >,
< lineOrdr=102, lineItem=71, lineQnty=80 >,
< lineOrdr=103, lineItem=27, lineQnty=250 > })

U14 = MODIFY(Line, (lineItem=47),
{ lineOrdr=lineOrdr, lineItem=lineItem, lineQnty=lineQnty+250 })

We wish to remark on some interesting aspects of these updates. Notice that update U1 is irrelevant to eleven of the seventeen derived relations. Of the remaining six derived relations—AvlbEast, AvlbCent, AvlbWest, FillEast, FillCent, and FillWest—none is autonomously computable since each requires more attribute values than are contained in the inserted tuples. On the other hand, U6 is irrelevant to fourteen and the remaining three —DistEast, AvlbEast, and FillEast—are all autonomously computable. Moreover, all three reside at the same regional office, hence no data need be transmitted from this site to any other; not even the update operation! By way of contrast, consider U7 which is very similar to U6. This time instead of combining regions 18 and 19 into 18, we combine 19 and 20 into 20. The update is irrelevant to eleven derived relations; of the remaining six, three are autonomously computable and three are not. The six derived relations are DistEast, AvlbEast, FillEast, DistCent, AvlbCent, and FillCent. Changes to the first three are autonomously computable as we can tell which tuples correspond to distributors in region 19 and hence should now be removed from these derived relations. On the other hand, changes to the latter three are not autonomously computable as we cannot determine from their present instances what tuples (corresponding to distributors which have had their region changed to 20) should be inserted.

Acknowledgements

The authors would like to thank the referees for their constructive and detailed comments which led us to improve the clarity and rigour of this paper.

References

[1] Serge Abiteboul and Victor Vianu. "Transactions and Integrity Constraints." In *Proc. of the 4th. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, (Portland, 1985), 193–204.

[2] Michel E. Adiba and Bruce G. Lindsay. "Database Snapshots." In *Proc. of the 6th. International Conference on Very Large Databases*, (Montreal, 1980), 86–91.

[3] D.S. Batory. "Modeling the Storage Architectures of Commercial Database Systems." *ACM Transactions on Database Systems*, Vol. 10, No. 4, (December 1985), 463–528.

[4] Philip A. Bernstein and Barbara T. Blaustein. "A Simplification Algorithm for Integrity Assertions and Concrete Views." In *Proc. COMPSAC 81*, (Chicago, 1981), 90–99.

[5] José A. Blakeley. *Updating Materialized Database Views.* Ph.D. Thesis, Department of Computer Science, University of Waterloo, 1987.

[6] José A. Blakeley, Neil Coburn, and Per-Åke Larson. "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates." In *Proc. of the 12th International Conference on Very Large Data Bases*, (Kyoto, 1986), 457–466.

[7] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. "Efficiently Updating Materialized Views." In *Proc. of the ACM SIGMOD International Conference on Management of Data*, (Washington, 1986), 61–71.

[8] O. Peter Buneman and Eric K. Clemons. "Efficiently Monitoring Relational Databases." *ACM Transactions on Database Systems*, Vol. 4, No. 3, (September 1979), 368–382.

[9] Robert W. Floyd. "Algorithm 97: Shortest Path." *Communications of the ACM*, Vol. 5, No. 6, (June 1962), 345.

[10] G. Gardarin, E. Simon, and L. Verlaine. "Querying Real Time Relational Data Bases." In *IEEE-ICC International Conference*, (Amsterdam, 1984), 757–761.

[11] Michael Hammer and Sunil K. Sarin. "Efficient Monitoring of Database Assertions." In *Supplement Proc. ACM SIGMOD International Conference on Management of Data*, (Austin, 1978), 159.

[12] Per-Åke Larson and H.Z. Yang. "Computing Queries from Derived Relations." In *Proc. of the 11th International Conference on Very Large Data Bases*, (Stockholm, 1985), 259–269.

[13] Per-Åke Larson and H.Z. Yang. "Query Transformation for PSJ-queries." In *Proc. of the 13th International Conference on Very Large Data Bases*, (Brighton, 1987), 245–254.

[14] Bruce Lindsay, Laura Hass, C. Mohan, Hamid Pirahesh, and Paul Wilms. "A Snapshot Differential Refresh Algorithm." In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, (Washington, 1986), 53–60.

[15] David Maier. *The Theory of Relational Databases.* Computer Science Press, Rockville, MD (1983).

[16] David Maier and Jeffrey D. Ullman "Fragments of Relations." In *SIGMOD'83 Proc. of Annual Meeting, Sigmod Record*, Vol. 13, No. 4, (December 1983), 15–22.

[17] Daniel J. Rosenkrantz and Harry B. Hunt III. "Processing Conjunctive Predicates and Queries." In *Proc. of the 6th International Conference on Very Large Data Bases*, (Montreal, 1980), 64–72.

[18] Hongzhi Yang. *Query Transformation.* Ph.D. Thesis, Department of Computer Science, University of Waterloo, 1987.