Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates

José A. Blakeley, Neil Coburn, and 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. 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 *PSJ*-expression.

This research was supported by Cognos, Inc., Ottawa under contract WRI 502-12, by the Natural Sciences and Engineering Research Council of Canada under grants No. A-2460 and No. A-9292, and by the National Council of Science and Technology of México (CONACYT).

Authors' address: Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1 Canada.
Electronic mail: {jablakeley,ncoburn,palarson}@waterloo.csnet.

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. 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, doing so after every update operation appears extremely wasteful and would probably be unacceptable, both from a performance and a cost point of view.

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

- determine the conditions under which the update U has no effect on the derived relation E_i, regardless of the database instance. In this case, the update U is said to be *irrelevant* to E_i

- if the update U is not irrelevant to E_i, then determine the conditions under which E_i can be correctly updated using only 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, U is said to be *autonomously computable* over E_i.

The update 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 [TK 78]. In current systems there is a one-to-one correspondence between conceptual relations and stored relations, that is, each conceptual relation exists as a separate stored relation (file). 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 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 a paper by Larson and Yang [LY 85].

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 [BC 79] 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 [HS 78] 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 considered by Gardarin et. al. [GSV 84], 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. On the other hand, 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 [AL 80, BLT 86, L 86].

2. Notation and Basic Assumptions

We assume that the reader is familiar with the basic ideas of relational databases as in Maier [M 83]. A *derived relation* is a relation instance resulting from the evaluation of a relational algebra expression over a database instance. 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. 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 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 query 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 **A** will often be referred to as the *visible* attributes of the derived relation. For simplicity, we assume that each relation of **R** occurs only once in the relational algebra form of the *PSJ*-expression, that is, we do not allow self-joins. 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
$V(E,d)$	The relation resulting from evaluating the relational expression E over the instance d of D

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

DELETE $(R_u, \mathbf{R}_D, C_D)$
Delete from relation R_u all tuples satisfying condition C_D, where C_D is a selection condition over the relations $\mathbf{R}_D$, $\mathbf{R}_D \subseteq D$

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

Every DELETE or MODIFY operation must specify the set of tuples from R_u to be updated. Selecting the set of tuples to be deleted from or modified in R_u can be seen as a query to the database. In the same way as derived relations, these "selection queries" are restricted to those defined by *PSJ*-expressions. For the update operation DELETE$(R_u, \mathbf{R}_D, C_D)$, the set of tuples to be deleted from R_u is selected by the *PSJ*-expression $E_D = (\alpha(R_u), \mathbf{R}_D, C_D)$. Similarly, for the operation MODIFY $(R_u, \mathbf{R}_M, C_M, \mathbf{F}_M)$, the set of tuples to be modified in R_u is selected by the *PSJ*-expression $E_M = (\alpha(R_u), \mathbf{R}_M, C_M)$.

The set $\mathbf{F}_M$ is assumed to contain an update expression for each attribute in R_u. We restrict the update expressions in $\mathbf{F}_M$ to unconditional functions that can be computed "tuple-wise". Unconditional means that the expression does not include any further conditions (all conditions are in C_M). Tuple-wise means that, for any tuple in R_u selected for modification, the value of the expression can be computed from the values of the attributes of that tuple alone. The update expressions are computed simultaneously, that is, all "new" values are computed from "old" values. The type of expressions we have in mind are simple, for example, $H := H + 5, I := 5$. Further details are given in section 4.3. 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. If the attributes from which the new values are computed, are from a relation R_v, $R_v \neq R_u$, then it is unclear which tuple in R_v should be used to compute the new values.

All attribute names in the base relations are taken to be unique. We also assume that all attributes have 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. For Boolean expressions, 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 $\forall C$; similarly for existential quantification. If we need to explicitly identify which variables are quantified, we write $\forall_X (C)$ 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.

3. Basic Concepts

Detecting whether an update operation is irrelevant or autonomously computable involves testing whether or not certain Boolean expressions are valid, or equivalently, whether or not certain Boolean expressions are unsatisfiable.

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

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. However, for a restricted class of Boolean expressions, polynomial algorithms exist. Rosenkrantz and Hunt [RH 80] 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 $x\ op\ y + c$ or $x\ op\ c$, where $op \in \{=, <, \leq, >, \geq\}$, x and y are variables, and c is a (positive or negative) constant. Each variable is 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 [LY 85] developed an algorithm whose running time is $O(n^2)$. However, it does not handle expressions of the form $x\ op\ y + c$ 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. Full details are given in [BCL 86].

An expression not in conjunctive form can be handled by first converting it into disjunctive normal form and then testing each conjunct separately. Several of the theorems in sections 4 and 5 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 $C_1 C_2 \vee (\neg C_1)(\neg C_2)$.

The concepts covered by the three definitions below were introduced in Larson and Yang [LY 85]. As they will be needed in sections 4 and 5 of this paper, we include them here for completeness.

Definition: Let C be a Boolean expression over the variables $x_1, x_2, \ldots, x_n$. The variable x_i is said to be *nonessential* in C if

$$\forall x_1,\ldots,x_i,\ldots,x_n, x_i' \left[C(x_1,\ldots,x_i,\ldots,x_n) = C(x_1,\ldots,x_i',\ldots,x_n) \right].$$

Otherwise, x_i is *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, variable H is nonessential in the following two conditions:

(1) $(I > 5)(J = I)((H > 5) \vee (H < 10))$, and
(2) $(I > 5)(H > 5)(H \leq 5)$.

Definition: Let C_0 and C_1 be Boolean expressions over the variables $x_1, x_2, \ldots, x_n$. The variable x_i is said to be *computationally nonessential* in C_0 with respect to C_1 if

$$\forall x_1,\ldots,x_i,\ldots,x_n, x_i' \left[C_1(x_1,\ldots,x_i,\ldots,x_n) C_1(x_1,\ldots,x_i',\ldots,x_n) \Rightarrow (C_0(x_1,\ldots,x_i,\ldots,x_n) = C_0(x_1,\ldots,x_i',\ldots,x_n)) \right].$$

Otherwise, x_i is *computationally essential* in C_0. □

If a variable x_i (or a subset of the variables $x_1, x_2, \ldots, x_n$) is computationally nonessential in C_0 with respect to C_1, we can correctly evaluate the condition C_0 without knowing the exact value of x_i. That is, given tuple $t = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ where the full tuple (including x_i) is known to satisfy C_1, we can correctly determine whether or not t satisfies C_0. This can be done by determining a surrogate value for x_i as explained in Larson and Yang [LY 85].

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

Definition: Let C be a Boolean expression over the variables $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m$. The variable y_i is said to be *uniquely determined* by C and $x_1, \ldots, x_n$ if

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

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 in Larson and Yang [LY 85]. 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: 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: Let $E = (\mathbf{A}, \mathbf{R}, C)$ be a derived relation and let $\mathbf{A}_e$ be the set of all attributes 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. □

Note that $\mathbf{A}^+$ is the maximal set of attributes for which values can be reconstructed for *every* tuple of E.

4. Detecting Irrelevant Updates

This section considers irrelevant updates. We deal with insertions, then deletions, and finally the most difficult case, modifications. First we define what it means for an update to be irrelevant [BLT 86].

Definition: Let d denote an instance of the base relations D and d' the resulting instance after applying the update operation U to d. Let E be a derived relation. The update operation U is *irrelevant* to E if $V(E, d') = V(E, d)$ for all instances d and d'. □

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

4.1. Irrelevant Insertions

An insert operation INSERT(R_u, T) is irrelevant to a derived relation if none of the new tuples will be visible in the derived relation. Note that this should hold regardless of the state of the database. The following theorem was proven in [BLT 86] and is included here for completeness only.

Theorem 1: The operation INSERT(R_u, T) is irrelevant to the derived relation $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, if and only if $C[t]$ is unsatisfiable for every tuple $t \in T$. □

4.2. Irrelevant Deletions

A delete operation is irrelevant to a derived relation if none of the tuples in the derived relation will be deleted. We have the following theorem.

Theorem 2: The operation DELETE(R_u, $\mathbf{R}_D$, C_D) is irrelevant to the derived relation $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, if and only if the condition $C_D \wedge C$ is unsatisfiable.

Proof: Let $\mathbf{B} = \mathbf{R} \cup \mathbf{R}_D = \{R_{i_1}, R_{i_2}, \ldots, R_{i_k}\}$. $\mathbf{B}$ is called the *combined base* of the derived relation and the delete operation. We first show that we can extend the base of E to $\mathbf{B}$ without affecting the resulting derived relation in any way. Without loss of generality, we can assume that $\mathbf{R}_D - \mathbf{R} = \{R_{i_1}\}$ so that $\mathbf{R} = \{R_{i_2}, R_{i_3}, \ldots, R_{i_k}\}$. Let t be a tuple in the Cartesian product $r_{i_2} \times r_{i_3} \times \cdots \times r_{i_k}$ (the base before adding R_{i_1}). If t satisfies C, then $t[\mathbf{A}]$ (the projection of t onto $\mathbf{A}$) will be visible in the derived relation, otherwise it will not. Extending the base to $r_{i_1} \times r_{i_2} \times \cdots \times r_{i_k}$ may give rise to a number of "copies" of t in the extended base. The copies differ only in the attributes of R_{i_1}. Since

$$\alpha(C) \subseteq \bigcup_{j=2}^{k} \alpha(R_{i_j})$$

then $\alpha(R_{i_1}) \cap \alpha(C) = \emptyset$. Hence, if t satisfies C, then all its copies will satisfy C. Similarly, if t does not satisfy C, then none of its copies will satisfy C either. The projection onto $\mathbf{A}$ will finally reduce all copies of t to a single tuple, exactly $t[\mathbf{A}]$. This proves that extending the base of E does not change the resulting derived relation. In the same way, we can show that extending the base of the delete query $E_D = (\alpha(R_u), \mathbf{R}_D, C_D)$ to $\mathbf{B}$ has no effect. We now complete the proof of the theorem.

(Sufficiency) Let t be a tuple over the combined base $\mathbf{B}$ and assume that t satisfies C. Then $t[\mathbf{A}]$ is visible in the derived relation. If $C_D \wedge C$ is unsatisfiable, then t cannot at the same time satisfy C_D. Hence $t[\mathbf{A}]$ will not be deleted from the derived relation.

(Necessity) Assume that $C_D \wedge C$ is satisfiable. We can then construct an instance of each relation in $\mathbf{B}$ such that deleting a tuple from r_u, ($R_u \in \mathbf{B}$), will indeed change the derived relation. 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 $X^0 = \langle x_1^0, x_2^0, \ldots, x_l^0 \rangle$ such that $C[X^0] C_D[X^0]$ is *true*. We now construct one tuple t_{i_j} for each relation $R_{i_j} \in \mathbf{B}$. The attribute values of t_{i_j} are assigned as follows: if the attribute occurs in $\alpha(C) \cup \alpha(C_D)$, assign it the corresponding value from X^0, otherwise assign it an arbitrary value in its domain, the minimum value, for example. We now have a database instance where each relation, as well as the Cartesian product $r_{i_1} \times r_{i_2} \times \cdots \times r_{i_k}$, contains one tuple. The tuple in the Cartesian product obviously satisfies C and hence the derived relation also contains one tuple. It also satisfies C_D and hence the relation r_u, will be empty after the deletion operation has been performed. Therefore, evaluating E over the new instance of the database will result in the empty set. This proves that the stated condition is necessary. □

Example: Consider two relations $R_1(H, I, J)$ and $R_2(K, L)$ and the following derived relation and delete operation:
$E = (\{H, L\}, \{R_1, R_2\}, (J = K)(K > 10)(I = 5))$ and DELETE$(R_1, \{R_1\}, (J < 5)(I < 10))$.

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

$$\forall\, I, J, K \quad \neg \left[(J = K)(K > 10)(I = 5)(J < 5)(I < 10) \right].$$

This is equivalent to proving that

$$\nexists\, I, J, K \quad \left[(J = K)(K > 10)(I = 5)(J < 5)(I < 10) \right]$$

which can be simplified to

$$\nexists\, I, J, K \left[(J = K)(I = 5)(K > 10)(K < 5) \right].$$

The condition $(K > 10)(K < 5)$ can never be satisfied and therefore the delete operation is irrelevant to the derived relation. □

4.3. Irrelevant Modifications

Modifications are 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 and, furthermore, all the attributes visible in the derived relation remain unchanged

Some additional notation is needed at this point. Consider a modify operation MODIFY $(R_u, \mathbf{R}_M, C_M, \mathbf{F}_M)$ and a derived relation $E = (\mathbf{A}, \mathbf{R}, C)$. Let $\alpha(R_u) = \{B_1, B_2, \ldots, B_l\}$. For simplicity we will associate an update expression with every attribute in R_u, that is, $\mathbf{F}_M = \{f_{B_1}, f_{B_2}, \ldots, f_{B_l}\}$ where each update expression is of the form $f_{B_i} \equiv (B_i :=$ <arithmetic expression>). If an attribute B_i is not to be modified, we associate with it a *trivial update expression* of the form $f_{B_i} \equiv (B_i := B_i)$. If the attribute is assigned a fixed value c, then the corresponding update expression is $f_{B_i} \equiv (B_i := c)$. The notation $\rho(f_{B_i})$ will be used to denote the right hand side of the update expression f_{B_i}, that is, the expression after the assignment sign. The notation $\alpha(\rho(f_{B_i}))$ denotes the set of variables mentioned in $\rho(f_{B_i})$. For example, if $f_{B_i} \equiv (B_i := B_j + c)$ then $\rho(f_{B_i}) = B_j + c$ and $\alpha(\rho(f_{B_i})) = \{B_j\}$.

By substituting every occurrence of an attribute B_i in C by $\rho(f_{B_i})$ 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 $B_i \in \alpha(R_u) \cap \alpha(C)$.

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

$$C_B(\mathbf{F}_M) \equiv \bigwedge_{B_i \in \alpha(R_u)} (\rho(f_{B_i}) \le U_{B_i})(\rho(f_{B_i}) \ge L_{B_i})$$

Theorem 3: The modify operation MODIFY $(R_u, \mathbf{R}_M, C_M, \mathbf{F}_M)$ is irrelevant to the derived relation $E = (\mathbf{A}, \mathbf{R}, C)$, $R_u \in \mathbf{R}$, if and only if

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

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

The proof is omitted due to space limitations; for the full proof see [BCL 86]. The following example illustrates the theorem.

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

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

$$\text{MODIFY}\ (R_1, \{R_1\}, (H > 20), \{(H := H + 5), (I := I)\}) .$$

Thus the condition given in Theorem 3 becomes

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

which can be simplified to

$$\begin{aligned} &\forall H, I, K \\ &\Big[(H > 20)(H \leq 25) \\ &\Rightarrow (\neg((H > 10)(I = K))) \\ &\qquad \wedge (\neg((H > 5)(I = K))) \\ &\quad \vee (H > 10)(I = K)\Big] . \end{aligned}$$

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. □

5. Autonomously Computable Updates

If an update operation is not irrelevant to a derived relation, then some data from the database is needed to correctly update the derived relation. The simplest case is when 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 current state of the derived relation and the information contained in the update expression.

Definition. Consider a derived relation E and an update operation U, both defined over base relations D. Let d denote an instance of D before applying U and d' the corresponding instance after applying U. The effect of the operation U on E is said to be *autonomously computable* if there exists a function F_U such that

$$V(E, d') = F_U(V(E, d))$$

for all database instances d and d'. Apart from the information in U itself, the only other data required by F_U must be contained in the current instance of E. □

5.1. Insertions

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

A. given a tuple $t \in T$ we can correctly decide whether t will satisfy the selection condition C (regardless of the database instance) 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.

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 tuples elsewhere in the database.

Theorem 4A: Let $E = (\mathbf{A}, \mathbf{R}, C)$ be a derived relation and t a tuple to be inserted into relation R_u, where $R_u \in \mathbf{R}$. Whether or not t will create an insertion into the derived relation is guaranteed to be autonomously computable if and only if one of the following holds:

I. $\mathbf{R} = \{R_u\}$

or

II. $\mathbf{R} \neq \{R_u\}$ and all the variables of $C[t]$ are nonessential and the current instance of E is non-empty.

Proof: (Sufficiency)

Case I: Since $\mathbf{R} = \{R_u\}$ then $\alpha(C) \subseteq \alpha(R_u)$. Hence, $C[t]$ can be completely evaluated, i.e. will yield *true* or *false*.

Case II: The fact that all variables in $\alpha(C[t])$ are nonessential guarantees that $C[t]$ will evaluate to the same value regardless of the values assigned to those variables. Since the current instance of E is non-empty, the Cartesian product of all relations in $\mathbf{R} - \{R_u\}$ will contain at least one tuple. Combining t with a tuple from this Cartesian product gives a tuple with fixed values for all variables in $\alpha(C)$ and the condition can be evaluated. Whatever the values of the attributes in $\alpha(C[t])$ are, the condition will always evaluate to the same truth value. Hence, whatever the current instance of the database the decision will always be the same.

(Necessity) Assume that whether or not t will create on insertion into the derived relation is autonomously computable but that neither of the two cases holds. Since the second case contains three conditions, three possibilities arise:

- $(\mathbf{R} \neq \{R_u\})$ and $(\mathbf{R} = \{R_u\})$. This is obviously a contradiction.
- $(\mathbf{R} \neq \{R_u\})$ and there exists some variable, $x \in \alpha(C[t])$, which is essential in $C[t]$. Without loss of generality we can assume that x is the only variable in $\alpha(C[t])$. This means that there exists two different values x' and x'' such that $C[t, x']$ is *true* and $C[t, x'']$ is *false*. In the same way as in the proof of Theorem 2, we can construct two different instances d' and d'' of D. Instance d' is constructed from x' and instance d'' from x'', such that, except for the given values of x, all the corresponding attribute values agree. In both instances relation R_u is empty and every other relation in D consists of a single tuple. Hence,

  $$V(E, d') = V(E, d'') = \emptyset$$

  Now insert tuple t into relation R_u. Since $C[t, x']$ is *true*, $V(E, d')$ must have a new tuple inserted, whereas $V(E, d'')$ will not, as $C[t, x'']$ is *false*. Consequently, whether or not insertion of t will affect the derived relation depends on the existence of tuples not seen in the derived relation.
- $(\mathbf{R} \neq \{R_u\})$ and the current instance of E is empty. There are two situations which would cause E to be empty; either no tuple in the Cartesian product of the base relations satisfies C or one of the base relations is empty. If $R_v \in \mathbf{R}, R_v \neq R_u$, is empty then even if $C[t]$ is *true* ,t will not cause an insertion into E. Consequently, whether or not the insertion of t will affect the derived relation depends on the existence of tuples in the other relations in the base of E, that is, on the existence of tuples outside the derived relation. □

Theorem 4B: Assume that a tuple in T has been shown to cause the insertion of a new tuple into the derived relation. The values of all visible attributes in the new tuple are guaranteed to be autonomously computable if and only if $\mathbf{A} \subseteq \alpha(R_u)$.

Proof: (Sufficiency) Obvious.

(Necessity) Without loss of generality we can assume that $\mathbf{A} - \alpha(R_u)$ contains only one attribute $x \in \alpha(R_i), R_i \neq R_u$. Assume that $t \in T$ causes the insertion of a new tuple. To insert the new tuple into the derived relation we must determine the value of x. Even if the value of x is uniquely determined by the attribute values of t, this is not sufficient. The value of x must correspond to the x value in some tuple in R_i, and the existence of such a tuple cannot be guaranteed without checking the current instance of the relation R_i. □

5.2. 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 5: The effect on the derived relation $E = (\mathbf{A}, \mathbf{R}, C)$ of the update operation DELETE $(R_u, \mathbf{R}_D, C_D)$, $R_u \in \mathbf{R}$, is guaranteed to be autonomously computable if and only if every attribute in $\alpha(C_D) - \mathbf{A}$ is computationally nonessential in C_D with respect to C.

Proof: (Sufficiency) If the variables in $\alpha(C_D) - \mathbf{A}$ are all computationally nonessential, we can correctly evaluate the condition by assigning surrogate values.

(Necessity) Without loss of generality we can assume that $\alpha(C_D) - \mathbf{A}$ consists of a single attribute x. 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. Each of t_1 and t_2 can now be extended into an instance of D. 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. The decision depends on the value of attribute x which is not visible in the derived relation. Hence the decision cannot be made without additional data. □

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

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

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

The attributes in the set $\alpha(C_D) - \mathbf{A} = \{H, I\} - \{J, K\} = \{H, I\}$ must be computationally nonessential in C_D with respect to C in order for the deletion to be autonomously computable. That is, the following condition must hold:

$$\forall H, I, H', I', J, K$$

$$\left[(I = J)(H < 20)(I' = J)(H' < 20) \Rightarrow ((I = 20)(H < 30)) = ((I' = 20)(H' < 30))\right].$$

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 $J = I = I'$. Because $I = I'$, the conditions $I = 20$ and $I' = 20$ are either both *true* or both *false*, and hence the consequent will always be satisfied. Therefore, the variables H and I are 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 variables H and I.

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

E: 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 $t_1 = (10, 15)$ 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, I that make $C[t_1] = \mathit{true}$, are valid surrogate values, say $I = 10, H = 19$. We can then evaluate C_D using these surrogate values, and find that $(10 = 20)(19 < 30) = \mathit{false}$. Therefore, tuple $t_1 = (10, 15)$ should not be deleted from E. Similarly, for $t_2 = (20, 25)$ we obtain $C[t_2] \equiv (I = 20)(H < 20)$. Surrogate values for H and I that make $C[t_2] = \mathit{true}$ are $I = 20$ and $H = 19$. We then evaluate C_D using these surrogate values and find that $(20 = 20)(19 < 30) = \mathit{true}$. Therefore, tuple $t_2 = (20, 25)$ should be deleted from E. □

5.3. Modifications

Deciding whether modifications can be performed autonomously is more complicated than for either insertions or deletions. In general, a modify operation may generate insertions into, deletions from, and modifications of existing tuples of the derived relation as a result of updating a base relation. Proving that an update is autonomously computable can be divided into the following four 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 autonomously compute which tuples in the derived relation should be modified. Call this the modify set.

C. Prove that we can autonomously compute 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**.

For each of these four steps we have found both sufficient and necessary conditions. Lack of space prevents us from including the results here, full details are available in [BCL 86]. The conditions are of a similar nature to those of previous theorems, but are somewhat more complicated. As before, they can be tested at run-time and without accessing the database. Again, the concepts of computationally nonessential, uniquely determined, and satisfiability play a crucial rôle in these conditions.

6. Conclusions

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 class of update operations considered is 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 mean that it will always 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 main 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.

If an update is not autonomously computable some additional data may be required. An open problem is to determine the minimal amount of additional data required from the database, and how to retrieve it efficiently.

References

[AL 80]
Adiba, M., and Lindsay, B.G., "Database Snapshots," Proc. 6th International Conf. on Very Large Databases, (1980), 86-91.

[BCL 86]
Blakeley, J.A., Coburn, N., and Larson, P.-Å., "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates," Technical Report CS 86-17, University of Waterloo (1986).

[BLT 86]
Blakeley, J.A., Larson, P.-Å., and Tompa, F.W., "Efficiently Updating Materialized Views," Proc. ACM SIGMOD International Conf. on Management of Data, (1986), 61-71.

[BC 79]
Buneman, O.P., and Clemons, E.K., "Efficiently Monitoring Relational Databases," ACM Trans. on Database Systems, 4, 3 (1979), 368-382.

[GSV 84]
Gardarin, G., Simon, E., and Verlaine, E., "Querying Real Time Relational Data Bases," IEEE-ICC International Conference (1984), 757-761.

[HS 78]
Hammer, M. and Sarin, S.K., "Efficiently Monitoring of Database Assertions," Supplement, Proc. ACM SIGMOD International Conf. on Management of Data, (1978), 38-48.

[LY 85]
Larson, P.-Å. and Yang, H.Z., "Computing Queries from Derived Relations," Proc. 11th International Conf. on Very Large Databases, (1985), 259-269.

[L 86]
Lindsay, B.G., et.al., "A Snapshot Differential Refresh Algorithm," Proc. ACM SIGMOD International Conf. on Management of Data, (1986), 53-60.

[M 83]
Maier, D., *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

[RH 80]
Rosenkrantz, D.J. and Hunt, H.B. III, "Processing Conjunctive Predicates and Queries," Proc. 6th International Conf. on Very Large Data Bases, (1980), 64-72.

[TK 78]
Tsichritzis, D.C. and Klug, A. (eds.), "The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Database Management Systems," Information Systems 3 (1978).