Understanding the implications of view update policies†

Claudia Bauzer Medeiros
Frank Wm. Tompa

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

ABSTRACT

Database views are traditionally described as unmaterialized queries, which may be coincidentally updatable according to some fixed criteria. One of the problems in updating through views lies in determining whether a given view modification can be correctly translated by the system. To define an updatable view, a view designer must be aware of how an update request in the view will be mapped into updates of the underlying relations. Furthermore, because of side effects, the view designer must also be made aware of the effects of underlying updates back into the view.

To address this problem, we present a general algorithm that predicts the effects of arbitrary mapping policies. Given an update policy, this algorithm indicates whether a desired update will, in fact, occur in the view and describes all possible side effects it may have, documenting the conditions under which they occur. The algorithm subsumes the results obtained by other view design tools, and generalizes their use to encompass a larger class of views.

1. Introduction

In the relational model, views are defined as single-relation images of queries. When updating through views, the updates must be mapped into the underlying database, preserving the database consistency, and reflecting the desired change in the view. Researchers have proposed ways of choosing the "appropriate" update mapping, having thereby restricted the domain of updatable views. Considerable effort has been spent on defining general view update translators (e.g., by treating views according to their complements [BAN81], or as components of a boolean algebra [HEG84]), and on defining views together with the updates they support (e.g., by treating them as abstract data types [TUC83]). The need for fast update processing motivates research on the complexity of update algorithms [COS83], efficient methods of checking integrity constraints [SIM84], or maintenance of materialized updatable views [SHM84]. In other related research, update semantics have been analyzed independently of the existence of views [NIC82, FAG83].

The so-called *view update problem* [e.g., CAR79, FUR79, DAY82, KEL82, HEG84, KEL85] centers around characterizing underlying updates that correctly reflect a change in the view . However, the definition of a "correct" translation may vary with the users' intentions, and even depend on the database's state at update time. This has forced researchers to restrict the set of views that can be updated to those where only unambiguous changes can be specified. As a consequence, only a limited set of updates through views have traditionally been supported.

This paper presents a design tool for handling the update translation problem. It can be used to process both general update translators and specific view update policies, therefore unifying previous view design approaches. It consists of an algorithm which allows a view designer to analyze and document the meaning of any update request by indicating the associated update translation. As a result, most update requests lose their ambiguity, since the update's effect is stated by means of the associated translation. By taking into consideration all possible valid database states, this algorithm predicts whether the desired update does, in fact, occur in the view, and whether it will result in additional modifications to the view. Determination of view interference (i.e., when updates to a view modify other views) is achieved by a simple extension of the algorithm.

Unlike all other approaches, the aim of the method presented here is to allow database system implementers to liberalize the translation policies,

† This work was supported in part by grant A9292 from the Natural Sciences and Engineering Research Council of Canada and by scholarship 200 398/80 from Conselho Nacional de Desenvolvimento Cientifico e Tecnologico - CNPq, Brasil.

rather than to remain restricted to a small set of valid interpretations. Among other things, the designer can specify several types of actions to be taken when exceptions occur, treat underlying changes as atomic operations distinct from the transaction <single insertion, single deletion>, and correctly process both traditional and non-standard relational views (such as lossy views). Furthermore, whereas most update validation tools restrict themselves to dealing exclusively with functional dependencies [e.g., LIN78, ARO78, DAY82, KEL82, KEL85], the algorithm proposed also extends the set of constraints manipulated to encompass join dependencies. These liberalizations come as a consequence of the algorithm taking into account factors which are considered unacceptable by most update validation tools (e.g., the contribution to the view of "dangling tuples" or of attributes that have been eliminated by means of projections). Some of these factors have been previously suggested as necessary for a more complete analysis, but they are consistently ignored, since it is claimed that they add too many degrees of freedom to the problem, and are not amenable to efficient analysis [e.g., CAR79, KEL82].

The reader is assumed to be familiar with basic relational database terminology, as described for example in Ullman's textbook [ULL82]. Only static update analysis will be considered (i.e., database state transitions are not discussed), and it is assumed that view access mechanisms (such as authorization and serialization of operations) are handled elsewhere by the system.

This paper is organized as follows: Section 2 contains basic concepts, notation and assumptions; Section 3 describes the validation algorithm proposed; Section 4 describes the rules for forming the tableau used by the algorithm to predict side effects; Section 5 summarizes results about the algorithm's correctness and complexity; Section 6 contains two examples of policy validation; and Section 7 presents the conclusions.

2. Notation and assumptions

Let R = $\{R_i <\{A_{ij}\},\{C_{ik}\}>\}$ be a relational database scheme, where $\{A_{ij}\}$ and $\{C_{ik}\}$ represent the attributes and constraints of R_i. In this paper, $\{C_{ik}\}$ is considered to be composed of a set of minimal keys (representing key functional dependencies) and at most one join dependency, which will be represented by a simple, full, typed template dependency.

Template dependencies [SAD82] consist of a template with a set of hypothesis rows $(w_1 \cdots w_s)$ and a conclusion row (w_{s+1}). Each row is composed of a set of symbols representing attribute values. A template dependency can be considered as a logic clause $w_1 \wedge \cdots \wedge w_s \rightarrow w_{s+1}$. This is interpreted as indicating that if all hypotheses hold (i.e., the tuples they represent exist in the relation), then the relation must also contain the tuple(s) described by the conclusion row. A dependency is *typed* if no symbol appears in more than a column; a dependency is *full* if every symbol in the conclusion row also appears in some hypothesis. A dependency is *simple* if at most one symbol is repeated for any column. Sagiv [SAG85] has shown that simple full typed template dependencies correspond to join dependencies.

Because of the complexity inherent in addressing inference problems for template dependencies (such as described by Fagin *et al.* [FAG83a]), it is assumed that templates cannot span relation schemes, and that there is at most one simple template dependency for any underlying relation scheme. There is, however, no limitation on the number of keys allowed.

A view V is defined as a pair V = (R, q_v) and a set of update rules $\{u_v\}$. q_v is the view generating function, which, applied to R, will present the desired query image. It is assumed that q_v is restricted to projections, selections and natural joins over R. Thus, for example, views formed by aggregation operations will not be considered.

Instead of actual tuples, the validation algorithm manipulates symbolic expressions describing sets of tuples in a relation. Each expression consists of a string of *parametric* and *placeholder* variables, where each variable stands for an attribute in the corresponding scheme. Parametric variables (denoted by uppercase characters) stand for specific attribute values at execution time, and placeholder variables (in lowercase characters) represent all other attribute values. The symbol $\{t\}_i$ denotes a set of tuples in scheme R_i.

Example:

Let R_1 = (**P**arent, **C**hild, **D**ept) have as one instance {(John,Mark,CS), (John,Sue,CS), (Mary,Paul,Math)}. Let (John,*,CS) refer to the first two tuples of this instance. The related symbolic expression is $\{t\}_1$=(PcD), i.e., the set of tuples with specific (parametric) attribute values for attributes Parent and Department to be given at execution time. (Note that this expression also describes the tuple (Martin,*,English), etc.)

For brevity, all references to the updating or the existence of a *template row* actually refer to the updating or the existence of the *tuples* described by that row. For instance, the statement "(ABC) is inserted if (ABc_2) exists" means that "given a relation r with attributes A, B and C, the tuple described by (ABC) is inserted if this relation contains tuples described by $\exists\, c_2 \mid (ABc_2) \in r$".

3. Description of the validation algorithm

The algorithm can be used to validate single insertions, deletions, and changes. Usually, requests for tuple replacement are considered to be equivalent to a transaction of type <single deletion, single insertion> [e.g., NIC82], but as remarked by Keller [KEL82] this is not always realistic. Since one of the aims of the approach presented here is that of placing as few restrictions as possible on the set of policies a user can implement and analyze, flexibility is increased by treating change requests as yet another type of atomic operation.

The input to the algorithm is a pair $<R,q_v>$, representing view formation and consistency information, and a single update rule u_v to be validated. Each update rule can be considered to be a procedure of the form *<view update desired, {underlying transformations}>*, which is activated at execution time by an update request. *View update desired* describes the update operation to be performed and plays the role of procedure header. The set *underlying transformations* is the executable body, defining how to translate the update operation into underlying updates.

Each *underlying transformation* is specified as the 4-tuple

<Op, tuples affected, R_i, exception action>.

Op is the type of underlying operation: *De* (deletion), *In* (insertion), or *Ch* (change);

tuples affected is denoted by $\{t\}_i$ or $\{\{t,t'\}_i\}$ (the latter format is used when the operation is a change: $\{t\}_i$ replaces $\{t'\}_i$). For underlying insertions, the user can only check the results of inserting a tuple at a time, i.e., $\{t\}_i$ is an expression with parametric variables only. For deletions and changes, the user can specify operations over sets of tuples.

R_i is the underlying relation being updated;

exception action indicates the type of action to be taken if the operation violates a constraint, and may be "force" or "cond"(conditional), which is the default.

Conditional updates are performed only if no other modifications to the underlying database are needed; this is the action implicitly meant by most mapping analyses when specifying the set of views where an update can occur. For example, *<De*,(PcD),R_i,cond> means that all tuples in R_i that match the expression (PcD) for the values in P and D should be deleted, as long as this does not cause deletion of further tuples. Forcing updates is accomplished by adding, deleting or changing additional tuples so as to maintain the database consistency.

Previous researchers have assumed that valid view deletions are those that are translated into deletions of the underlying relation tuples, and valid view insertions are those that are translated into either underlying insertions, or replacements of null by non-null values in some underlying tuples [e.g., LIN78, DAY82]. These assumptions are again liberalized, by allowing any combination of underlying updates, as long as only one type of operation is specified for each relation. Thus it is possible to effect a deletion or a replacement by an underlying set of insertions.

Example:

If R_1=<(Name,Dept), {Name→Dept}> contains the tuple (Jones,CS), a request for conditional insertion of (Jones,Math) is rejected, since it cannot be performed without deleting the tuple (Jones,CS). A request for forced insertion of the same tuple, however, can be accepted under the assumption that it provides information on the real world which is "more correct" than the database's contents. The result of such an insertion would then be the *replacement* of the existing tuple by (Jones,Math). This type of interpretation was first suggested by Fagin, Vardi and Ullman [FAG83].

The input is initially processed to discard some forced insertions or changes that are inherently ambiguous. This situation occasionally arises when the relation to be updated is subject to functional dependencies that are entangled with a template dependency. For such special cases, these constraints force conflicting data to appear in the database, and further information would be needed in order to disambiguate the result of an update. This type of situation can be recognized in polynomial time in the size of the template dependency, and is dealt with at length elsewhere [MED85].

The central part of the validation algorithm is based on a process similar to tableaux chases. This part – called the q_v/chase process – consists of evaluating the update policy, using the tuples $\{t\}_i$ indicated in *{underlying transformations}*, so as to derive all possible side effects on the view. First, all underlying updates necessary to maintain the consistency of each relation are simulated. The result of performing this simulation, denoted by *Op*($\{t\}_i$,R_i,*exception*), is stored in the *underlying modification table*. Next, for each underlying update, the checking of side effects is performed by investigating what are the individual contributions of the updates described in the underlying modification table. This means that, for the set of transformations $\{(Op_i,\{t\}_i,R_i,exception)\}_{i=1..k}$, this check must be executed using the entries in the underlying modification table corresponding to the isolated contribution of ($\{t\}_1$,$\{t\}_2 \cdots$ and $\{t\}_k$) to the view. In particular, if a given set of underlying transformations is composed of both underlying deletions and insertions, the algorithm explicitly takes the deletions into account

when computing which tuples may be inserted in the view (since the deleted tuples cannot contribute to insertions in the view).

Example:

Consider $q_v = R_1 \bowtie R_2$, where R=$\{R_1$<AB,{}>, R_2<BC,{B→C}>}, with extensions r_1={(a_1 b_1), (a_1 b_2), (a_1 b_3), (a_3 b_3)}, r_2={(b_1 c_1), (b_2 c_1)}. Suppose the user policy implements changing view tuple ($a_1b_1c_1$) into ($a_3b_3c_3$) by deleting (a_1 b_1) from r_1 and inserting (b_3 c_3) in r_2, which actually achieves the desired change in the view. If the deletion of (a_1 b_1) is analyzed separately, it is seen that it causes the deletion of ($a_1b_1c_1$) from the view. The effect of inserting (b_3 c_3) in r_2 must also be considered by itself, as it inserts ($a_1b_3c_3$) in the view as well.

Each underlying operation is thus processed separately. This is done by creating a sequence of tableaux for each modified relation, as explained in the next section. In each sequence, every tableau is obtained from the previous one by application of one operator from q_v, simulating actual projections, selections and join operations over real tuples. The first tableau of the sequence for R_i contains the set of entries of the underlying modification table corresponding to $Op(\{t\}_i,R_i,exception)$. Unlike the usual concept of tableaux, a distinction is only made between user-supplied variables and system-supplied variables, with blanks where the relation does not contain the corresponding attribute.

The output is a set of symbolic expressions, representing all tuples that may be deleted, replaced or inserted by the policy. Additional output information is provided by the *database state table*, which is filled during execution of the algorithm. It records the database state description for which each underlying update initially occurs, and keeps track of changes in variables for each row of the tableau. Each symbolic output expression is associated with a sequence of entries in this table, describing the underlying database state necessary for obtaining the corresponding update outcome.

Example:

Let $q_v = \sigma_{A>10}(R_1 \bowtie R_2)$, R=$\{R_1$<AB,{}>, R_2<BC,{B→C}>}. Given the symbolic tuples $\{t\}_1$='ab', and $\{t\}_2$='Bc', the result of the q_v/chase process would be $\{t\}_{12}=A_1BC_1$: that is, this tuple appears in the view if 'ab' is in r_1, 'Bc' is in r_2 and the database state descriptions hold. Besides indicating what are the database state descriptions for updating R_1 and R_2, the table entry for $\{t\}_{12}$ contains:

[b=B] (determined by the join operator);

[c=C_1] (the functional dependency determines that the attribute must have a specific value, since $B \rightarrow C$ and the symbol B represents a particular value);

[a=A_1>10] (determined while executing the selection operator). A parametric value A_1 is assigned to this variable to indicate that the selection delimits the range of the corresponding attribute. Both A and C are assigned indices, indicating that the corresponding parametric variables are not directly specified by the user.

4. Creation of the underlying modification table by replacement rules

$Op(\{t\}_i,R_i,exception)$ is a set of expressions that describes the set of tuples inserted, deleted and changed as a result of following the rules that simulate the effect of an update Op on R_i. Essentially, the template that represents R_i is replaced by a set of rows that describe all inserted and deleted tuples in the underlying relation.

In the following pages, pairs of expressions of the form (t,[t']) will denote that {t} replaces {t'}, and the expression [t'] by itself denotes deleted tuples. The symbol w_i denotes the i-th row of a template, and $\bar{w}_i$ denotes the same row after any symbol substitution has been performed.

If the exception action is "cond" the update cannot imply any other updates, and thus

$In(\{t\}_i,R_i,cond)=\{t\}_i$;

$De(\{t\}_i,R_i,cond)=[\{t\}_i]$;

$Ch(\{t,t'\}_i,R_i,cond)=(\{t\}_i, [\{t'\}_i])$.

The rules that follow refer therefore to the "force" exception action, as it is the only case where underlying updates other than those explicitly specified by the user may be generated.

4.1. R_i is not being updated

The template is replaced by w_{s+1}, which indicates no change to the relation and yet allows it to participate in view formation.

4.2. R_i is not subject to any integrity constraints

Insertion or deletion of $\{t\}_i$ does not violate any constraints, and no extra updates are needed (i.e., the only key consists of all attributes in R_i).

Therefore, $In(\{t\}_i,R_i,force)=\{t\}_i$;

$De(\{t\}_i,R_i,force)=[\{t\}_i]$;

$Ch(\{t,t'\}_i,R_i,force)=(\{t\}_i, [\{t'\}_i])$.

4.3. R_i is subject to a template dependency

Insertion of tuples is equivalent to modifying the hypotheses $w_1 \cdots w_s$ [NIC82]. Insertion effects must be tested separately for each hypothesis w_j, replacing it by the inserted tuple and applying substitution inference rules to the whole template, as described by Sadri and Ullman [SAD82]. The substituted conclusion rows $\{(\bar{w}_{s+1})_j\}$ indicate which additional tuples *may* need to be inserted to keep the clause valid, and the remaining hypothesis rows

$\bar{w}_{k,k\neq j}$ indicate the conditions under which insertion of $\{(\bar{w}_s+1)_j\}$ is needed.

Thus, $In(\{t\}_i,R_i,force) = (\{t\}_i \cup \{(\bar{w}_s+1)_j\})$.

Example:

Let R_1<ABC,{A→→B|C}>. The template before replacement and after (ABC) replaces each row is

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_1	b_2	c_1

A	B	C
A	B	C
A	b_2	c_2
A	b_2	C

A	B	C
A	b_1	c_1
A	B	C
A	B	c_1

Forced insertion of (ABC) therefore results in a table with three rows: (ABC) itself, and the substituted conclusion rows above: (Ab_2C) and (ABc_1). The database state table entries associated with insertion of (Ab_2C), for instance, are:

$$\exists\, b_2,c_2 | (Ab_2c_2) \in r_1.$$

For deletions, $\{t\}_i$ initially replaces the conclusion row, and again substitutions are applied to the template. If the substituted conclusion row also appears as some hypothesis $\bar{w}_j$, then satisfaction of the dependency is automatically achieved, because $\bar{w}_j$ is also deleted. If no such $\bar{w}_j$ exists, then another row may need to be eliminated to perform the deletion. This is chosen to be the first substituted hypothesis row that contains at least one parametric variable, denoted by $\bar{w}_{1d}$; the remaining modified hypothesis rows indicate the conditions under which deletion of $\bar{w}_{1d}$ is needed. Thus, $De(\{t\}_i,R_i,force)=[\{t\}_i \cup \bar{w}_{1d}]$. In the previous example, forced deletion of (ABC) results in a tableau with rows [ABC] and $[(Ab_1C)]=\bar{w}_{1d}$.

For changes, each template row w_j is duplicated, so that $w_j \leftarrow (w_j,[w_j])$. The conclusion row thus formed is replaced by $(\{t\}_i,[\{t'\}_i])$, and substitution rules are applied to each duplicated row. If the conclusion is not repeated in some hypothesis row, the change is propagated as indicated by the first row $\bar{w}_{1c}=(\bar{w}_k,[\bar{w}'_k])$ where $\bar{w}_k \neq \bar{w}'_k$. This is followed by any additional insertions that may be caused by appearance of $\{t\}_i$ and $\bar{w}_{1c}$. Therefore, $Ch(\{t,t'\}_i,R_i,force)=(\{t\}_i,[\{t'\}_i]) \cup (\bar{w}_{1c},[\bar{w}'_{1c}]) \cup In(\{t\}_i,R_i,force) \cup In(\bar{w}_{1c},R_i,force)$. For the same example, request of forced change (ABc,[AB′c]) initially generates the substituted template

A	B	C	[A	B	C]
(A	b_1	c_1,	[A	b_1	c_1])
(A	B	c_2,	[A	B′	c_2])
(A	B	c_1,	[A	B′	c_1])

where only the last two rows are considered for the underlying modification table, since the first row corresponds to "no change". Since the second row describes the same modification as the conclusion row, there is no need for propagating the change. Therefore, the entry for $Ch(\{t,t'\}_i,R_i,force)$ in the underlying modification table consists exclusively of the conclusion row indicated.

4.4. R_i is subject to non-trivial key dependencies

If there are only functional dependencies, the template corresponds in fact to a single symbolic expression describing the inserted tuple (or the set of changed tuples).

Deletion of tuples cannot violate a functional dependency, so $De(\{t\}_i,R_i,force)$ is calculated as in 4.3 if there is a template dependency, or as in 4.2 if not. A functional dependency X→Y may be violated by forcing either insertions or changes involving parametric values in X and Y. For every such dependency, both forced insertions and changes must be accompanied by modification of other tuples' values in the attributes in Y. This is achieved by specifying a row (w,[w′]) such that w and [w′] match $\{t\}_i$ in the parametric attributes corresponding to X, and have different values for the attributes in Y. Once the pairs {(w,[w′])} are generated, indicating preliminary operations necessary to solving any conflicts arising from functional dependencies, the set of rules in 4.3 for template dependencies is applied to determine which other tuples may need to be inserted or deleted to maintain the template dependency. Finally, any conflicts that may still arise (because the new insertions generated by the rules 4.3 for template dependencies may disagree with existing tuples over functional dependencies) are solved by deleting the offending tuples from the relation.

Example: (having functional dependencies only)
Let R_1<ABC,{A→BC; C→BA}>. Forced insertion of (ABC) may violate either dependency. Thus, the expressions generated are (ABC,[AB′C′]), for the tuple whose A-value matches A, and (ABC,[A′B′C]), for the tuple whose C-value matches C.

Consider again the same scheme R_1. The forced change (aBc,[aB′c]), for instance, transforms all B-values of all tuples in the relation to the value B. Another forced change, (ABc,[AB′c]), is transformed by chase into (ABC_1,[AB′C_1]) because A is a key and has a parametric value. This change modifies only the B-value of the tuple whose A-value is given by A, and whose C-value may be unknown to the user but is uniquely determined by the system.

Example: (Of having two types of dependency)
Let R_1<ABC,{A→→B|C, BC→A}>, whose template is

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_1	b_2	c_1

Forced deletion of (ABC) generates the rows (ABC) and (Ab_1C)=$\bar{w}_{1d}$.

Forced insertion of (ABC) is processed as follows:

1. insertion of (ABC) may require deletion of some existing tuple (A'BC), because of the functional dependency;
2. deletion of (A'BC) alone may not be possible because of the template dependency. Therefore, this deletion is processed using rules 4.3 for template dependencies, which causes tuples of the form $\overline{w}_{1d}$=(A' b_1 C) to be deleted for all values of b_1 such that $\exists\, c_2 |$ (A'Bc_2) $\in r_1$;
3. (ABC) is then processed as an insertion for template dependencies (rules 4.3) and the insertions $\{(\overline{w}_s+1)_j\}$ generated are (Ab_2C) and (ABc_1);
4. These additional insertions (in 3.) must be examined to see if the updated relation violates the functional dependency. In other words, existing tuples of the form (A'' b_2 C) and (A'' B c_1) (where $A'' \neq$ A) may have to be deleted.
5. Finally, (A'' b_2 C) and (A'' B c_1) alone may not be deletable because of the template dependency. These deletions are therefore processed against the template dependency, and the result is

$\forall\ b_1,c_2$ [(A'' b_1 C)] - deleted if (A'' b_2 C) is deleted (in 4.) and (A'' b_1 c_2) $\in r_1$;

$\forall\ b_1,c_2$ [(A'' b_1 c_1)] - deleted if (A'' B c_1) is deleted (in 4.) and (A'' B c_2) $\in r_1$. The final set of entries in the underlying modification table is

A	B	C	[A	B	C]
(A	B	C	[A'	B	C])
			[A'	b_1	C]
A	b_2	C	[A''	b_2	C]
			[A''	b_1	C]
A	B	c_1	[A''	B	c_1]
			[A''	b_1	c_1]

and the corresponding database state table records the conditions for each update.

5. Analysis of the validation algorithm

Theorem: The validation algorithm correctly indicates all possible results for any user update policy, and the underlying state necessary for obtaining each result.

This theorem is proved by reliance on the following major results [MED85]:

- The template replacement rules that generate $Op(\{t\}_i,R_i,exception)$ create a minimal set of underlying updates that perform the underlying operation Op over relation r_i, while maintaining the relation's consistency.
- Each result row of the q_v/chase process describes a possible effect that the underlying updates may have on the view, and the initial database state for which that effect occurs.
- Testing the q_v/chase process with the inputs $\{\{t\}_1 \cdots \{t\}_k\}$ is sufficient to describe all possible effects on the view caused by the underlying operations in the policy.

The number of comparisons in executing the q_v/chase process for each relation scheme R_i is polynomial, and is bounded by the chase of functional dependencies performed when building the underlying modification table and when processing tableau sequences. Forced insertions and forced changes are the most expensive operations, since they may require the type of housekeeping activity indicated in 4.4. Details on the complexity of this algorithm, and on methods to optimize its execution time and eliminate redundant rows are described by Medeiros [MED85].

6. Illustrative examples

Consider the relations

R_1=<(Part,Qty,Wt), {Part→(Qty,Wt)}>,

R_2=<(Part, Agent, Co.),

{*[(Part,Agent), (Part,Co.), (Agent,Co.)]}>,

and $q_v=\Pi_{(Part,Agent,Co.,Wt)}R_1 \bowtie R_2$. The templates built from this specification are

Part	Qty	Wt
p_1	q_1	w_1

Part	Agent	Co.
p_2	a_1	c_2
p_2	a_2	c_1
p_3	a_1	c_1
p_2	a_1	c_1

(1) Test the update (In,(P,A,C,W)), for the policy {(In,(P,A,C),R_2,force), (In,(P,Q,W),R_1,force)}

1.1. Inserting (**P,A,C**) into r_2

Replacing the first row of the template by (P,A,C) results in the consequent row (PAc_1); replacing the second row results in (Pa_1C); replacing the third row results in (p_2AC). The initial tableau is therefore

Part	Qty	Agent	Co.	Wt
p_1	q_1			w_1
P		A	C	
P		A	c_1	
P		a_1	C	
p_2		A	C	

Joining $R_1 \bowtie R_2$ causes the rows to be merged and requires (p_1=P), for joining the first row to the three subsequent rows; and (p_2=p_1), for joining the first and the last rows. This information is stored in the execution condition table to show that the join will take place only if tuples satisfying these conditions exist in r_1. Chasing the functional dependency over the modified tableau rows changes variables (q_1,w_1) to parametric values, since the key (Part) is parametric. Finally, applying the view projection on (*Part,Agent,Co.,Wt*) yields the rows {(PACW_1), (Pa_1CW_1), (PAc_1W_1), (p_1ACw_1)}.

This result, in conjunction with the associated database state description, indicates that forced insertion of (P,A,C) into r_2 might have the side-effect of adding other tuples to the view. For example, if (P,A,C) = (Tire,Smith,FIAT), then tuple (Tire,Smith,FIAT,5kg) is inserted in the view if (Tire,10,5kg) is in r_1 and (Tire,Smith,FIAT) is not already in r_2. Tuple (Tire,Smith,VW,5kg) will also appear if (Tire,10,5kg) is in r_1 and r_2 already contains (Tire,Jones,VW) and (Window,Smith,VW).

1.2. Inserting **(P,Q,W)** into r_1

The result row is (Pa_1c_1W),[(Pa_1c_1W')]). The associated conditions are interpreted as follows: if, for instance, (P,Q,W) = (Tire,20,10kg), and tuple (Tire,15,8kg) already exists in the relation, then all agents and companies associated with (Tire, 8kg) in the view will become associated with (Tire, 10kg). If the part (Tire) does not exist in r_1, and (Tire,Jones,VW) exists in r_2, then the tuple (Tire,Jones,VW,10kg) is inserted into the view.

1.3. Analysis of the two previous output stages shows that the specified tuple (P,A,C,W) = (Tire,Smith,FIAT,10kg) is inserted in the view, although others may be inserted as well. Therefore, if these side effects are not desired, the view should not support this type of insertion.

(2) Test the update (De,(P,A,C,W)), for the policy {(del,(P,A,C),R_2,cond), (del,(P,Q,w),R_1,force)}.

2.1. Isolated deletion of **(P,Q,w)** is transformed into deletion of (P,Q,W_1) because of the dependency Part→(Qty,Wt). The result is that if **Q** is, in fact, the quantity value for part P, all tuples of the form ($Pa_1c_1W_1$) will be deleted, where P→W_1. For example, if (P,Q,W) = (Wheel,15,30kg), then all view tuples containing (Wheel,*,*,30kg) will be deleted. If **Q** is not the quantity stored for **P**, nothing will be deleted from the view as a result of a deletion in R_1.

2.2. Deleting **(P,A,C)** from r_2 creates the substituted template

Part	Agent	Co.
P	A	c_2
P	a_2	C
p_3	A	C
P	A	C

This deletion cannot always be performed, since no hypothesis row is automatically deleted as well. Being a conditional update, it can only be executed if $\forall\ p_3,a_2,c_2$, ((PAc_2) $\notin r_2 \vee$ (Pa_2C) $\notin r_2 \vee$ (p_3AC) $\notin r_2$). If this condition is met, ($PACW_1$) is deleted from the view. For instance, if (P,A,C) = (Wheel,Black,FIAT), and if the relation contains the tuples (Wheel,Black,BMW), (Wheel,Brown,FIAT) and (Clutch,Black,FIAT), then (Wheel,Black,FIAT) cannot be deleted.

2.3. Analysis of these output stages shows that not only might the tuple *not* be deletable (due to the execution conditions for the updates), but if it were deleted, others might also disappear from the view.

7. Conclusions

This paper presents an algorithmic approach to validating updates through views at design time, showing how the effects of arbitrary update policies can be systematically derived in an error-free way. For a particular update request in a view, the system's translation rules can be applied to produce the update transformation from which the algorithm will generate not only the set of conditions under which the update will take place, but also the set of all possible side effects that might result from such an update.

The framework supported by the algorithm allows it to generalize and encompass the results of other validation tools, when applied to the limited set of problems they were designed to handle. For example, as shown by Medeiros [MED85], policies essentially equivalent to those of Furtado *et al.* [FUR79] (using a hierarchical view model), Dayal and Bernstein [DAY82] (employing trace graphs) or Keller [KEL85] (using the structural model for defining views created on BCNF relations), are all expressible in this framework.

The algorithm is also capable of handling policies which have so far been explicitly disallowed or not even considered. This helps in the formulation of new design policies and also enlarges the set of views which can be updated, including many which, in general, have been considered to be query-only views (e.g., allowing reference to sets of attributes not visible in the external schema, as well as modification of such attributes).

Rather than restricting the designer to a system-wide update translation, this approach can be used to document and validate an appropriate policy (or even a set of policies) for each view. For example, it extends the concept of operational views as proposed by Spyratos [SPY82], and can be applied to views implemented as abstract data types as proposed by Tucherman *et al.* [TUC83], validating the translations which are part of the view definition. For such cases, the designer can compare the outcomes of different policies, and choose to implement the policies where side effects are less likely to occur.

Finally, with very simple extensions, the algorithm can also be used for detecting the effects of view interference (instead of (R, q_v, u_v), the input used is {(R,{q_v}$_j$,u_v)}, involving repeated executions of the q_v/chase process for each view generated by {q_v}$_j$), checking the invariance of view complements [BAN81] (instead of q_v, the input uses q_{v^c}, the function that defines the complementary view), and

validating sequences of update operations (transactions), since they are composed of single insertions, deletions and changes. The results presented can be extended to monotonic view generating functions other than those exclusively composed of projections, selections and joins (by including the effects of other operators *after* creation of the underlying modification table using $Op(\{t\}_i, R_i, exception)$).

8. Bibliography

[ARO78] Arora,A.K. and Carlson,C.R., The information preserving properties of relational database transformations. *VLDB 1978*, 352-359.

[BAN81] Bancilhon,F. and Spyratos,N., Update semantics of relational views. *ACM TODS, 6(4), 1981*, 557-576.

[CAR79] Carlson,C.R. and Arora,A.K., The updatability of relational views based on functional dependencies. *COMPSAC 1979*, 415-420.

[COS83] Cosmadakis,S. and Papadimitriou,C.H., Updates of relational views. *PODS 1983*, 317-331.

[DAY82] Dayal,U. and Bernstein,P.A., On the correct translation of update operations on relational views. *TODS, 8(3), 1982*, 381-416.

[FAG83] Fagin,R. Ullman,J.D. and Vardi,M.Y., On the semantics of updates in databases. *PODS 1983*, 352-364.

[FAG83a] Fagin,R. Maier,D. Ullman,J.D. and Yannakakis,M., Tools for template dependencies. *SIAM Journal of Computing, 12(1), 1983*, 36-59.

[FUR79] Furtado,A.L. Sevcik,K.C. and Santos,C.S., Permitting updates through views of data bases. *Information Systems 4(2), 1979*, 269-283.

[HEG84] Hegner,S.J., Canonical view update support through boolean algebras of components. *PODS 1984*, 163-172.

[KEL82] Keller,A.M., Updates to relational databases through views involving joins., *2nd International Conference on Databases, Jerusalem, 1982.*

[KEL85] Keller,A.M., Sequence of update algorithms for translating view updates to database updates for views involving projections, selections and joins. *PODS 1985*, 154-163.

[LIN78] Ling,T.-W., Improving database integrity based on functional dependencies. *PhD thesis, Department of Computer Science, University of Waterloo, 1978.*

[MED85] Medeiros,C.M.B., A validation tool for designing database views. *PhD thesis, Department of Computer Science, University of Waterloo, 1985* (to appear).

[NIC82] Nicolas,J.M., Logic for improving integrity checking in relational databases. *Acta Informatica, 18(3), 1982*, 227-254.

[SAD82] Sadri,F. and Ullman,J.D., Template dependencies: a large class of dependencies in relational databases and its complete axiomatization. *JACM 29(2), 1982*, 363-372.

[SAG85] Sagiv,Y., On computing restricted projections of the representative instance. *PODS 1985*, 171-180.

[SHM84] Shmueli,O. and Itai,A., Maintenance of views. *SIGMOD 1984*, 240-255.

[SIM84] Simon,E. and Valduriez,P., Design and implementation of an extendible integrity subsystem. *SIGMOD 1984*, 9-17.

[SPY82] Spyratos,N., An operational approach to data bases. *PODS 1982*, 212-219.

[TUC83] Tucherman,L., Furtado,A.L. and Casanova,M.A., A pragmatic approach to structured database design. *VLDB 1983*, 219-231.

[ULL82] Ullman,J.D., Principles of database systems. *Computer Science Press, 2nd edition, 1982.*