A validation tool for designing database views that permit updates

Claudia Maria Bauzer Medeiros

Data Structuring Group
CS-85-44

November, 1985

A validation tool for designing database views that permit updates

Claudia Maria Bauzer Medeiros

Data Structuring Group — Department of Computer Science
University of Waterloo
Waterloo, Ontario, N2L3G1
Canada

ABSTRACT

This thesis presents a new approach to analyzing view update policies. Unlike all other approaches, the aim of the method presented is to liberalize update translations, so that database implementers need not be restricted to using a small set of valid interpretations (e.g., in the model underlying insertions can be used to delete view tuples).

The major contribution is in the presentation of a predictor algorithm that checks the validity of any update translation proposed by a view designer. This allows the designer to analyze and document the meaning of any update request by indicating the associated update translation as well as actions to be taken when exceptions occur. The output from executing the algorithm indicates whether the desired update is actually reflected in the view, and it describes all possible side effects to the proposed operation (as well as the database states for which each type of side effect can occur).

The algorithm subsumes the results obtained by other view design tools, and generalizes their use to encompass a larger class of views. Furthermore, updates that have traditionally been disallowed can be acceptably specified and algorithmically verified to have the intended effects on the view. Finally, whereas most update mappings have been analyzed in the presence of functional dependencies alone, the algorithm encompasses integrity constraints that include simple typed template dependencies as well.

Acknowledgements

I consider myself very fortunate in having had Frank Tompa as my supervisor, and am grateful for his encouragement, feedback and endless patience at all stages of my research. I also wish to thank the members of my thesis committee, Alan Adamson, Per Larson, Alberto Mendelzon and Ian Munro, for their comprehensive reading of my thesis and their insightful comments.

Special thanks should go to Jose Blakeley-Perez for having provided many comments which have resulted in a more readable final version of this thesis. Esmond Ng and Mark Mutrie must be commended for their courage and patience in having bravely withstood years of sharing an office with me.

The secretarial staff of the Department of Computer Science, especially Debbie Clermont and Anne Harris, deserve many thanks for having helped me jump over all bureaucratic obstacles. I would also like to thank all the people who made my stay in Waterloo a rewarding experience, due to their friendship and encouragement, in particular Nancy and Bruce Char, Kay and Neil Coburn, Linda Duxbury, Marta and Gaston Gonnet, Lorna Hall, Rolf Karlsson, Judy and Eva Kovacs, and Cathy and Howard Johnson.

These acknowledgements must include my parents, Ethel and Jorge Medeiros, who used love by remote control to keep me happy and secure.

Finally, I would like to acknowledge the financial support of CNPq - Brazil (Conselho Nacional de Desenvolvimento Cientifico), and of the Department of Computer Science of the University of Waterloo.

Table of Contents

Chapter 1

Introduction

1.1. Some common update problems

In order that a database reflects changes in the enterprise it is modelling, it must accurately support requests for modifications in the data. The effects of database updates have been analyzed under different perspectives, both at the logical level (e.g., design of the conceptual schema and path analysis) and the physical level (e.g., space allocation and file characteristics). Related research deals with authorization mechanisms, serializability and, in distributed databases, transaction recovery and data migration. Yet another concern is that of interpreting the meaning of an update request, which may involve natural language interpretation and formal semantic analysis.

Update operations must preserve the database integrity requirements (rules that describe all legally permissible extensions of a schema [DAY82]). At the logical level, integrity violation, as observed by Date [DAT83], can occur in two different contexts. The first — interference — concerns the fact that two independently correct applications can interact destructively; this is analyzed under concurrency models. The other — which is the main concern of this thesis — refers to the correctness of an isolated update request, given the database consistency criteria.

Update processing requires the establishment of mapping rules between the database design levels: operations requested at the external level are transformed into their conceptual schema equivalents and are then mapped into the desired internal level modifications. The problem lies in associating the database state change, as specified by the user, with a set of operations performed on the underlying database: the interpretation of the request must correspond to what the user intends.

This means that, given the old and new external states, and a set of rules for mapping the conceptual into the external level, one must be able to find the sequence of transformations that will cause the underlying relations to change, in order to obtain the desired final external state. In particular, the requested update should be made with no unrelated data changes. As an additional concern, the database final state must be semantically consistent (i.e., it must satisfy all integrity constraints). The sequence of underlying transformations that performs the external update requested is called the *translation* of the update request. A given translation algorithm will also be called a *mapping policy*, or an *update policy*.

Thus, the update problem consists of determining the update translation that 1) actually reflects the update; 2) causes as few side effects as possible; and 3) results in a final consistent database.

The discussion that follows will focus on these issues at the conceptual level for a relational database model. All the situations analyzed in this thesis rely on the assumption that appropriate procedures exist that will perform the translation from the conceptual into the internal level. Moreover, there will be no analysis of authorization concerns, and the existence of a concurrency control mechanism to serialize update operations is assumed.

1.2. Approaching database consistency

1.2.1. Updates to databases

Among the current database design models, the relational model has been, in general, the only one to be subjected to a theoretical study of update anomalies. Dayal and Bernstein's [DAY82a] extension of their theory of the updatability of relational views to the network model is one of the few formal treatments of another model.

Existing approaches to the update problem can be roughly classified into three levels: 1) preventing inconsistencies during the design stage, 2) checking for their presence at execution time, and 3) allowing them to exist for limited periods of time. Prevention of anomalies is implemented through imposing, at the design stage, a set of integrity constraints to be enforced, and limiting the ways in which the database components can be built (e.g., by projection), thereby defining the set of allowed changes as early as possible. One of the most restrictive (and commonly adopted) rules is that the only allowable updates are those on relations whose structures coincide with the structure of part of the stored representation. In the preventive approach, an update is allowed only if there is assurance that no constraint violation will occur. An example of this type of approach is Simon and Valduriez's [SIM84] extendible integrity subsystem.

Checking for potential inconsistencies is done at execution time, as it involves a dependence on the database state (one should not, for instance, insert already existing tuples) and on the actual values of the updated fields (range specifications should be obeyed). Update requests generate a series of tests on the contents of the database, to verify whether the update makes sense (given the present state), and whether the desired state will keep the database consistent. This is done by analyzing the scheme *extension* (an instance of a relation), together with all the integrity constraints imposed on it. Cremer and Domann's AIM system [CRE83] exemplifies the class of detection techniques.

Maintaining temporarily inconsistent databases is an approach similar to that of processing transactions pending commit and rollback decisions. The result of performing sets of updates is placed in temporary storage. An example of this type of approach is presented by Neumann and Hornung [NEU82]: the database is checked at certain time intervals, when the decision is taken as to which updates should be accepted and which rejected.

Update operations usually pass through the first two levels of control (prevention and checking), and most authors restrict themselves to analysis of these. The third approach however must also be considered, since some updates do not make sense if executed independently: they have to belong to a transaction composed of a set of operations. In such cases, consistency checks are meaningless before the transaction can complete. For instance, given the relations R_1:(Dept, Number_of_employees), and R_2:(Dept,Employee), when an employee is hired by the department, either R_1 or R_2 is updated first, but not both at once, so that a temporary inconsistency is allowed to occur.

Both prevention and detection techniques have been studied under static and dynamic models. Dynamic analysis of the validity of update operations depends on a database's history: the consistency criteria have to include decisions that involve the database's previous states, rather than only the present one. The work in this area has focused mainly on defining classes of dynamic constraints with the help of logic.

Static analysis, on the other hand, defines *a priori* rules for update consistency, without taking the state transition into account (although the rules can be enforced dynamically). Authors usually restrict themselves to static update analysis, even though some [FUR79, KEL82] also describe some of the problems which arise in the dynamic case. Vianu [VIA83], for instance, proposes an approach to investigate the effects of dynamic constraints on database evolution, interpreting temporal changes through inferring static constraints from the dynamic ones.

1.2.2. Updates through views

A relevant factor that contributed to intensifying research in the database update area was the appearance of the concept of *views*. The problem of database consistency arose early in the research concerning database design models and techniques. When analyzed under the relational model, this problem motivated the development of a theory of normal forms [COD70, FAG77, COD79]. With the appearance of large and distributed systems, there was a growing need to take individual applications into consideration. Thus, most of the recent literature concerning updates in databases refers to view implementations and discusses connected issues (such as independence, integration, interference and equivalence). Some researchers develop special tools to help define their view models (for example, the algebra of quotient relations of Furtado and Kerschberg [FUR77], and the structural model of El-Masri and Wiederhold [ELM80]); others take advantage of widely used formal notions, such as logic [FAG83] or boolean algebra [HEG84].

Paraphrasing Keller [KEL82], a view can be regarded as a temporary relation against which database requests may be issued. Views are thus images of the database as seen by queries, which in turn reflect what subsets of data are relevant for each user application.

Views are generated by means of operations (e.g., select) performed on the underlying relations. The composition of operations that form a particular view is often called a *view generating function*. Theoretical analyses of such functions contribute to the study of the preventive approach, in that they investigate how to limit the ways in which relations can be combined to form views. For instance, Carlson and Arora consider views whose functional definition is obtained by equijoins, projections and selection/restriction [CAR79], as well as projections and natural joins [CAR80]; Keller [KEL82, KEL85] compares views generated exclusively by projections against those generated by natural joins, selections and projections, on which the properties of El-Masri's structural model hold; Shmueli and Itai [SHM84] consider tree hierarchies of materialized views obtained by projections and equijoins.

The concept of views enhances the separation of applications (since most database transactions will not affect all the views), thereby providing some immunity from data reorganization. By the same token, the existence of different views helps the enforcement of data protection and privacy policies, besides simplifying the task of defining user interfaces. The coexistence of different views, however, raises several considerations as far as updates are concerned. If a view is created through joins of two relations, for instance, one must analyze which — and how many — tuples of each source relation must change, in answer to a view update request.

Typically, views do not have an independent physical existence in a database, but rather operate through a conceptual definition and the corresponding mapping into storage. The structure of a view is defined by a sequence of operations on the conceptual schema. Thus, any transaction requested by a view is actually performed on the conceptual schema extension. A view update request is translated through appropriate functions into a database update request. Once the change is effected, an inverse translation may be needed to show the user the result achieved.

Retrievals are always easier to handle than updates, since they only require mapping of the schema extension into the view extension. This can be done either by building the view extension for each query (applying the view definition to the schema extension) or by modifying the query (so that it contains information about the view definition), and applying it directly to the schema extension. Since no changes are requested, any mapping that correctly retrieves the desired information is acceptable.

Since several views may access the same information in different ways, it is possible that an update operation as requested by one view may cause a (potentially undesirable) side effect on what is seen by other views. Side effects can in fact occur whenever there is more than one user accessing the same data, but the problem is usually studied under the view framework. As remarked elsewhere [e.g., FUR79, DAY82, KEL82, CAS84], the *view update problem* consists therefore in choosing or defining view update translations which 1) achieve the desired view modification with 2) as few side effects as possible, and 3) do not violate integrity constraints.

Some of the problems that may occur while processing updates are:

a) non-unique translation: there may be more than one sequence of logical operations that will perform the update required (so that there must also be rules for choosing an appropriate one). This is a situation in which the new external state may correspond to several database states;

b) appearance of side effects: the new database state may not map back into the expected external state (e.g., extra tuples may have been created).

c) absence of correct translation: there is no consistent database state that maps into the new external state (constraint violation). The final state may violate global integrity constraints (that apply to the entire underlying database), or local integrity constraints (that apply to an external portion — or view — of the database). This may affect the application that requested the update or, in the more complex case, another application;

d) ambiguity of interpretation: there may be more than one correct interpretation for a given update request, resulting in several possible external states.

Example: Consider the database with relations R_1=(Employee,Department) and R_2=(Department,Manager), subject to the functional dependencies E→D and D→M. Let two views $V_1 = R_1 \times R_2 =$ (Employee,Department,Manager) = and $V_2 = \Pi_{EM} R_1 \times R_2 =$ (Employee,Manager) be defined for this database.† Consider the following instances of the relations, and the resulting views

Employee	**Dept**
Yang	Food
Fletch	Food
Powell	Toys
Hart	Sports
Hoyt	Cars

Dept	**Manager**
Food	John
Toys	Bill
Sports	Linda
Clothes	John

Empl	**Dept**	**Manager**
Yang	Food	John
Fletch	Food	John
Powell	Toys	Bill
Hart	Sports	Linda

Empl	**Manager**
Yang	John
Fletch	John
Powell	Bill
Hart	Linda

a) non-unique translation: A request for changing (Yang,Food,John) in V_1 to (Smith,Food,John) can be achieved either by changing (Yang,Food) to (Smith,Food) in R_1, or by inserting (Smith,Food) in R_1 and changing (Yang,Food) to (Yang,NULL) — where NULL denotes some marked null value.

b) side effects: A request for insertion of (Smith,Cars,Jane) in view V_1, which can be performed by inserting (Smith,Cars) in R_1 and (Cars,Jane) in R_2, but this will also insert (Hoyt,Cars,Jane) in the same view.

c) constraint violation: A request for insertion of the similar tuple (Smith,Clothes,Jane) in V_1 cannot be honored, even though it requires exactly the same type of translation procedure as (b), because there already exists a record (Clothes,John) in R_2, and this would violate the functional dependency D→M.

d) ambiguous interpretation: A request for deletion of (Yang,John) in view V_2 can be achieved either by deleting (Yang,Food) from R_1, or by deleting (Food,John) from R_2, or both. In the last two cases, this also deletes (Fletch,John) from the view.

† Throughout this thesis, the symbol ($\times$) stands for natural joins.

1.3. Formalizing the approaches to the view update problem

Casanova and Furtado [CAS84] divide formal approaches to the view update problem into two categories: treating the view as an abstract data type *versus* defining general translation procedures. This thesis proposes a new classification, which includes at least the following categories:

. the *functional mapping* approach, in which rules are set for defining functions for mapping an update requested at the external schema (or individual view) level into the underlying internal structure (and *vice-versa*);

. the *complement mapping* approach, in which these mappings are handled according to how they affect a view's complement;

. the *operational* approach, in which views are analyzed and hierarchically classified according to the operations that may be performed on them;

. the *abstract data type* approach, in which views are defined together with the operations they allow, and the translation for these operations;

. the *deductive database* approach, in which the database is considered as a first order theory, and updates are treated as changes in the theory sentences.

The two first categories belong to the group of general translation procedures; the third approach is a transition between general translation procedures and the abstract data type approach. The deductive approach does not necessarily take views into consideration, and does not fit the classification of [CAS84].

1.3.1. The functional mapping approach

The central characteristic of the research in this category is that it tries to specify desirable properties of view generating functions in order to determine which views can be updated. Integrity constraints are used to restrict the set of possible translations.

Even though prior attempts had been made to establish formal definitions of inter-schema mappings (e.g. Paolini and Pelagatti [PAO77]), one of the first comprehensive papers to formalize the notion of correct translatability is the one presented by Dayal and Bernstein [DAY78]. They propose theorems to define situations in which view updates (of type insertion, deletion or replacement) can be allowed in the relational model. The theorems are expressed in terms of view extensions, and the constraints are restricted to functional dependencies and keys. They analyze the process of mapping view updates to updates on the underlying extension, with the development of correctness criteria for these mappings. The central result is that an update on a view is translatable if it can be expressed by a *unique* series of update operations on the schema extension, so that there are no additional updates, no side effects, and semantic consistency is preserved. A restriction on this unique update sequence is that it can only be composed of updates of the same type as the original one (i.e. a sequence composed exclusively of insertions, or of deletions, or of replacement operations). As is the case in all other similar analyses, update operations are, in reality, requested on a relation. The authors later extended this work [DAY82] to include subset constraints, but the basic results were kept.

Furtado, Sevcik and Santos [FUR79] use the algebra of quotient relations [FUR77] (by means of which their conceptual view model is defined) to build the transformations between an update request and the corresponding conceptual modification. Their mapping functions are more complex than those of Dayal and Bernstein [DAY82], being computed through an iterative process: a view can be built from other views. Any update on it will be first translated into updates on its source views, and thereafter into the relations that originated these source views. This backtracking can, theoretically, go through an arbitrary number of levels, provided this stepwise process of view building is derived using just one operation of the quotient algebra per iteration. Some special composition processes (such as projection of a restriction) are treated using simplified mapping rules.

Masunaga [MAS84] provides general translation rules using the same iterative process of [FUR79]. He analyzes insertions and deletions in terms of the view definition function (which can be composed of a series of union, difference, select, project and product operators). Even though he describes different alternatives to a translation, his model is not concerned with integrity constraints. Unlike other approaches in this area, where integrity constraints help determine the "correct" translation, the translatability of an update depends uniquely on the view generating function.

Keller [KEL85] presents a set of algorithms that translate updates through views when these views can be expressed as a set of select operators, followed by a set of projects, followed by a natural join. The only constraints allowed are functional dependencies, and — like Dayal and Bernstein's [DAY82] updatability criteria — all relation keys and join attributes must be present in the view (in fact, all joins must be extension joins over relation keys). The criteria his algorithms must satisfy to be considered as adequate translations are: side effects cannot occur; each database tuple can only be affected once by a given update; and the translation must be minimal. His minimality requirements are that no unnecessary changes need occur, and that there cannot be insertions and deletions in a relation at the same time (they should be transformed into changes).

Brosda and Vossen [BRO85] describe general conditions for which an insertion or a deletion through a universal relation view can be accepted. Their results aim at preserving the correctness of the representative instance (see Chapter 6), and the constraints consist exclusively of functional dependencies. Furthermore, the tuples affected by the update cannot contain any nulls.

1.3.2. The complement mapping approach

Spyratos [SPY80] and, later, Bancilhon and Spyratos [BAN81] study update mappings according to the effect they have on view complements. A *view complement* is a second (virtual) view from which all the database information omitted from the first one can be deduced. Thus, any database state can be computed from a view and its complement. According to this definition, a view can be contained in its own complement (even though, in this case, the complement alone will characterize the whole database, and not just the information absent from the view).

The update operations that interest them are those that can be cancelled (i.e., undone). This means that there exists for every update an inverse operation, that returns the view to the original state. Furthermore, the composition of update operations must have the same result as applying them separately (allowing the same sort of backtracking as described in the previous section). The only updates they allow, among those that satisfy these requirements, are those that do not affect the view complement.

The central idea relies on the observation that a view update request has a unique translation if the request can be translated maintaining the complement invariant. The choice of an update policy is in fact dictated by first determining the desired complement, and then the update that leaves it invariant (which will then be unique under their model). In other words, once the complement is fixed, there is a single way of changing the view without modifying its complement.

An objection to this type of treatment is the fact that the process of obtaining a complement is not clear. Besides, the policy of predefining the complement so as to obtain a unique update mapping means that the user knows how each requested operation will affect the entire database. This, as observed by Fagin, Vardi and Ullman [FAG83], is very rarely the case.

Cosmadakis and Papadimitriou [COS83] extend the above work by analyzing the complexity of algorithms that implement updates according to the invariant-complement notion. They show that finding a minimum complement is NP-complete and demonstrate that for a view generated by projections, where the constraints are functional dependencies, finding a complement that renders an update translatable is polynomial in terms of the number of functional dependencies and size of the view extension. Other researchers who have considered this type of approach for studying update mappings are Hegner [HEG83, HEG84], who describes classes of views that maintain complement invariance, and Keller and Ullman [KEL84], who analyze properties characterizing view independence, where two views are independent if updating one of them does not modify the other.

1.3.3. The operational approach

This approach is connected with analyzing the extent to which database transformations preserve information, and ordering these transformations accordingly. Carlson and Arora [ARO78, CAR79] direct their research towards classifying views according to the operations they support. A central result of their formal model [CAR80] is that views that support insertions also support deletions (and thus that updatable views are those that support insertions). Their classification is used in analyzing conditions for schema modification and translation, as part of the view integration process.

Another example of this type of approach is found in Ling's thesis [LIN78]. Not only does it establish general translation rules for mappings between the conceptual and the external schemas — which would place his work in the functional mapping category — but it also defines and classifies general situations under which a view is updatable (*update viable*), when the constraints are functional dependencies. The classification separates unconditional or conditional update viability. Unconditional viability means the view can always accept the update proposed, since no undesired effects will occur — e.g., a view supports deletions unconditionally if its key is also the key of some underlying relation. Conditional viability means that the given view update may sometimes fail to be performed, depending on several factors including the database state when the update is requested and integrity constraints. Unconditional viability corresponds to Dayal and Bernstein's [DAY78] definition of *clean sources*: a relation is a clean source for a view update if updating the relation has exactly the desired effect on the view.

Whereas Carlson and Arora's and Ling's research classify updates according to their effect on the view itself, Klug [KLU78] classifies the type of view update translation according to its effect on *other* views. He shows there may be several mappings to translate a given view update request, and he suggests that update requests be associated with their translations, which he describes by a mixture of pseudo-code and first order logic clauses. His classification divides update effects into those with independent semantics (when the update has the exact effect desired) and dependent semantics (when other effects may occur). Dependent semantics may be strong, medium or weak, according to the type of effect the update has on other view tuples. Klug defines several correctness criteria for update translations and constructs formulas in predicate calculus that correspond to these criteria, but the validity of these formulas is undecidable in most cases.

One interesting aspect of Klug's model is that it considers both queries and updates to be operations that change view states, by defining the answer to a query as part of the view state. Thus, the state resulting from a query is composed of the view state at query time and the answer to the query. The state resulting from an invalid update request is the view state at update request time, together with the appropriate error message. Finally, the state resulting from a valid update is the view state after the update is performed.

The operational approach is also taken by Spyratos [SPY82], who defines views in terms of the operations performed therein: a view specification becomes a set of allowable updates on a subset of the database. The attributes and relations on which these updates operate are indicated indirectly: each possible tuple (and tuple combination) is assigned a different identifier, and an update is represented as a set of pairs [old identifier, new identifier]. This is a way of indicating how the database changes from one state into another. The database is divided into a set of view equivalence classes (named *definition sets*), according to the set of updates they support. A definition set is composed of all views on which a given (fixed) set of identifiers is mapped into the same set, after an update is performed. The complement of a class is the set of identifiers that do not change under those updates.

This means that the updates themselves are partitioned according to their interpretation (as to how they affect a class, and to which classes they can be applied). Allowable updates are those that do not interfere with the set partition. Similarly to the other authors in this section, Spyratos uses this classification to order views with respect to the operations allowed, besides defining properties such as equivalence and independence of views.

1.3.4. The abstract data type approach

This is the approach taken in designing databases in modules (usually views themselves) that are similar to abstract data types. Rather than forbidding updates because they do not obey pre-defined rules, this type of approach blocks invalid requests at the design stage by predefining which are the operations accepted by each module.

One of the first proponents of this approach is Clemons [CLE78], who points out that general mappings impose too many restrictions on views and proposes bypassing this problem by defining views together with all the operations they support, and their translations. Tucherman, Furtado and Casanova [TUC83] specify databases in terms of *modules*, which characterize all possible applications as abstract data types. These modules are composed of descriptions of the underlying schemes, constraints and operations, as well as the views allowed therein, and the operations attached to the views. The modules are nodes of a digraph, where the edges define a generation hierarchy: thus, new views can be formed from old ones, allowing operations permitted by their ancestors. One drawback to this type of approach is that there are no guidelines as to how appropriate translations should be determined.

The approach taken in this thesis to analyze update mappings can be classified as a mixture of concepts taken from both the abstract data type and the operational approaches.

1.3.5. The deductive database approach

This is the approach taken when databases are considered from the viewpoint of logic. A relational scheme is treated as a first order logic theory (a set of statements in first order logic), with a set of integrity constraints, and a database is a model of the theory's axioms. Atomic facts, integrity constraints and derivation rules are all expressed as sentences in logic without function symbols. The deductive database is considered to be a logic program that produces facts, and whose integrity constraints are program properties. Database updates correspond to program modifications that cannot destroy these properties. The two approaches to the problem — considering the database either as an interpretation of a first order theory or as a theory itself — are described at length by Gallaire, Minker and Nicolas [GAL84]. In the first case (which corresponds to conventional databases), queries and integrity constraints are formulas that are to be evaluated on an interpretation using the semantic definition of truth. The second case corresponds to a syntactic approach: queries and integrity constraints are theorems to be proved. The so-called *logic databases* correspond to this second viewpoint.

As Gallaire *et al.* point out, because of the combinatorial complexity of inferencing in logic databases, the application of logic to databases is usually restricted to the conventional framework. The work described in this section can be applied to either type of database, but only the conventional interpretation will be considered in this thesis. Expressing database integrity constraints as logic clauses is becoming increasingly popular, and is by no means restricted to logic models of databases [e.g., TUC83].

Nicolas and Yazdanian [NIC78] show how to insert or delete facts from relations expressed in terms of logic clauses, so as to obey their integrity constraints. Fagin, Ullman and Vardi [FAG83] (whose work was later extended by Kuper, Ullman and Vardi [KUP84]) attack the problem of semantics of updates in databases which are arbitrary theories. These theories are updated by modifying the set of sentences that characterize them. The authors consider how to change theories, and which set of operations to choose so as to keep the change minimal in size (since several different theories can represent the updated version). If there is more than one theory that accomplishes the update minimally (i.e., the update can result in different states), then the updated theory (new state) becomes the union of all such theories (of all possible alternative states).

The basic premise of their work relies on a novel approach: no updates should be rejected. (This type of procedure is also suggested in [NIC78], but is not analyzed at length therein.) Unlike the standard deductive approach, Fagin *et al.* even accept changes to the integrity constraints themselves. They also demonstrate that, even though the existence of views is blamed as the origin of all update anomalies, one can still have problems if the views are not considered.

The deductive approach is also used in aid of integrity checking. Several authors use this type of formalism to propose ways of simplifying the set of constraints that must be checked in order to ensure that a given database is consistent. Reiter [REI81] shows how to reject updates which violate type (i.e., domain) constraints. Nicolas [NIC82] simplifies integrity checking on some subsets of an updated database by instantiation of variables in the clauses that represent the integrity constraints. Kobayashi [KOB84] uses many-sorted predicate logic to show how to find validating procedures for update requests.

1.4. Update analysis as a byproduct of other design considerations

Section 1.3 reports work centered on database updates themselves. Updates have however also been analyzed in several other contexts, some of which are briefly mentioned in this section. Even though not the prime issue in discussion, they are studied in the database design and conversion processes. For example, equivalence of schemes, view integration and conversion, and schema separability are instances of ways in which the semantics of update operations is considered in the literature.

For a set of update requests corresponding to a view, analysis of the view scheme indicates which relations are affected (and thus how the database is perturbed). Given two view data models, and a set of update operations to be performed on both, the models are considered equivalent if affected in a similar way. This means, essentially, that both will have the same set of views at the end of the update execution. The operations requested by different views are examined in the context of equivalence of views. This issue is also raised in the view integration process [ELM80, MOT81], especially when these operations alter the database state [GAD82].

Wong and Katz [WON80] use update anomalies to detect inconsistencies in data models when converting a schema expressed in terms of the Entity-Relationship model into a relational or network model. Update independence motivates research into independence of schemes: if the relation schemes in a set

$\{R_1 \cdots R_n\}$ are independent, an update to a relation r_1 does not affect the other relations $\{r_2 \cdots r_n\}$. This type of analysis is performed, for instance, by Chan and Mendelzon [CHA83] and Honeyman and Sciore [HON83], who define different criteria for independence.

1.5. Purpose and organization of the thesis

In spite of the extensive work on database updates, most view update models are usually subject to very rigid rules. One such rule concerns the set of constraints under which view updatability models are studied. There are several descriptions of implemented systems that enforce a number of integrity constraints (e.g., from Stonebraker's [STO75] description of prevention of inconsistencies by query modification to Simon and Valduriez's [SIM84] enforcement of referential integrity and temporal assertions). Most formal analyses, however, assume that the only allowable integrity constraints are functional dependencies, since other types of constraints (such as multivalued dependencies) usually cause more ambiguity and are as such harder to analyze formally. Other basic assumptions which have limited the lines of research are: views are formed only through projection/joins of relations; the database (and views) must always correspond to a single relation; all attributes of the source relations are present (e.g. there is no projection); change operations are equivalent to an atomic transaction of type <single insertion, single deletion>; and updates must be requested only on single tuples of a single relation. The effects of relaxing some of these constraints, or of analyzing update consistency under a different set of constraints, have not yet been investigated.

Update translations into underlying operations are also subject to very strict rules — e.g., view deletions can only be translated to underlying deletions. Klug [KLU78] remarks on this and even shows how view deletions can be effected by underlying changes. He does not, however, pursue this line of thought, and in fact ignores changes and limits view insertions to insertions of a single tuple in each underlying relation (though he accepts that some view deletions can only be accomplished if several underlying tuples are deleted). Keller's recent proposal [KEL85] considers the possibility of coexistence of insertions and deletions in different relations, but still subject to very stringent criteria (e.g., minimality of change).

Another problem (whose existence has been, up to now, tacitly accepted and avoided, save by Fagin, Ullman and Vardi [FAG83] and Keller [KEL85]) is that of *forcing* the acceptance of *all* update requests. The reasoning behind this is that the new data may be more accurate than the information already in the database.

The rigidity of the framework used for analyzing update translations results in few views being considered to be updatable and in a paucity of alternative view update policies. This is aggravated by the assumption that, in most cases, there is no possibility of predicting which side effects may occur.

This thesis presents a new approach to analyzing view update policies under a static model. It assumes the existence of a view designer, whose main concern is to design views according to user specifications. Unlike all other approaches, the aim of the method presented here is to allow these designers to liberalize the translation policies, rather than to remain restricted to a small set of valid interpretations. Thus, they can take advantage of the system's characteristics to define any update mapping desired, most of which are forbidden by traditional approaches. An immediate consequence is that one can update views which have been until now considered as unable to support update operations.

This liberalization is supported by a predictor algorithm — the update validation algorithm — that checks the validity of any update translation proposed. It allows the view designer to analyze and document the meaning of any update request, by indicating the associated update translation, as well as actions to be taken when exceptions occur. As a result, update requests lose their ambiguity, since the update's effect is stated by means of the associated translation. Not only can the validation algorithm indicate whether the desired update is actually reflected in the view, but it also describes all possible side effects to the proposed operation, as well as the underlying database states in which each type of side effect can occur. This algorithm is a tool to be used by the database designer and, especially, by those responsible for designing views.

Besides enhancing the set of valid update translations, the algorithm extends the traditional framework of previous analyses. Not only does it consider individual insertion and deletion requests, but also validates sets of such requests, and treats tuple replacements as atomic operations different from insertions or deletions. Furthermore, whereas most update validation tools restrict themselves to dealing exclusively with functional dependencies [e.g., LIN78, ARO78, KEL85, BRO85], the algorithm proposed also extends the set of constraints manipulated to encompass the class of join dependencies, which will be processed as typed template dependencies [SAD82]. 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].

Since the algorithm can be used to process both general update translators and specific view update policies (see Section 1.3), it provides a much needed unifying framework under which previous view design approaches can be treated.

The outline of the thesis is as follows. In order to allow a better understanding of the validation method proposed, Chapter 2 gives a summary of some tools that are used in subsequent chapters. Chapter 3 describes the *update validation algorithm* and the interpretation of its symbolic output. Chapter 4 contains a formal analysis of the algorithm, proving its correctness and analyzing its complexity. Chapter 5 shows how the present approach generalizes and encompasses specific view design tools proposed by other researchers. Chapter 6 contains a summary of the thesis, and directions for future work.

Chapter 2

Tableaux and template dependencies

In the chapters that follow, a combination of modified tableaux and template dependencies is used as the means of representing update mappings, interpreting queries and deriving side effects of view updates. For this reason, this chapter contains a brief description of these concepts. Most of the material on tableau manipulation is based on Maier's book [MAI83]. The last section of this chapter describes basic assumptions used throughout the thesis.

2.1. Relations and schemes

A *relation scheme* R_i is a set of attribute names $\{A_{i_j}\}$, each of which corresponds to a domain of possible values, represented as $dom(A_{i_j})$. A *relational database scheme* is a set of relation schemes, R=$\{R_i\}$. A *tuple* defined on scheme R_i is a function that maps each attribute in R_i to a value in the attribute's domain. A *relation* r_i on R_i is a finite set of tuples defined on R_i. A *state* of a database scheme is a function that maps every relation scheme R_i to a relation r_i on R_i. The set of constraints C for the scheme R defines the legal database states: a state is legal if all relations in the database satisfy every constraint in the set C. An *instance*, or *extension*, of a database is a set of relations in one legal state of the database scheme.

2.2. Tableaux

Tableaux are traditionally employed in connection with *lossless decomposition* of relations, under the universal relation instance assumption. A relation r decomposes losslessly into scheme R=$\{R_1 \cdots R_n\}$ if $r = \Pi_{R_1}(r) \times \cdots \times \Pi_{R_n}(r)$. This expression is called a project-join mapping.

Among other applications, tableaux are used as a means of representing project-join mappings, deriving properties of relation schemes, and representing special classes of queries. Different forms of tableaux appear, for instance, when determining equivalence, optimization and modification of queries [MAI83, chapter 11], establishing dependency satisfaction [GRA82], or deriving implications of dependencies [MAI84].

In standard applications, a tableau is considered as a matrix that represents a relation. Instead of actual values, the matrix rows contain *distinguished* variables (usually denoted by subscripted a's), and *non-distinguished* variables (denoted by subscripted b's). Each tableau row represents one or more tuples in the corresponding relation. Tableau variables do not appear in more than one column, and each column can have at most one distinguished variable.

Let R_i be a relation scheme with attributes $\{A_{i_j}\}$, T be a tableau, $\{\xi\}_i$ be the set of variables appearing in T and $D = \bigcup_j dom(A_{i_j})$. Let $\xi_j \in \{\xi\}_i$ represent a variable appearing in the column corresponding to A_{i_j}. A *valuation* for elements in tableau T is a mapping σ from $\{\xi\}_i$ to D such that $\sigma(\xi_j) \in$ dom(A_{i_j}). Valuations are extended to sets of rows and to the entire tableau, as follows. Let w be a row in T, of the form $w = (\xi_1 \xi_2 \cdots \xi_n)$. The valuation $\sigma(w)$ is given by $\sigma(w) = (\sigma(\xi_j) \cdots \sigma(\xi_n))$; the tableau valuation $\sigma(T)$ is given by $\{\sigma(w) |\ w$ is a row in T}.

Example: Applying valuation $\sigma(a_1)=1$, $\sigma(a_2)=3$, $\sigma(b_1)=2$, $\sigma(b_2)=2$ to the tableau on the left results in the tableau valuation at the right:

T(A_1	A_2)	σ(A_1	A_2)
a_1	b_1	1	2
b_2	a_2	2	3

This valuation can be interpreted as "the relation whose extension is {(1,2),(2,3)} can be described by the tableau T." Thus, a tableau T for scheme R_i can be interpreted as a function on relations with scheme R_i. If w_d is a row composed exclusively of distinguished variables, *not necessarily in T*, and r_i is an instance of R_i, then

$$T(r_i) = \{\sigma(w_d) | \sigma(T) \subseteq r_i\}.$$

This means that if a valuation σ takes every row in T to a tuple in r_i, then $\sigma(w_d)$ is in $T(r_i)$.

The concept of tableaux as mapping functions is used in special cases by algorithms for determining equivalence of schemes. These algorithms are based on defining a tableau for each relation scheme, reducing these tableaux and comparing their characteristics. Tableau reducing operations rely on the concept of *covering*, or *subsumption*. Let w_1 and w_2 be rows in a tableau T for scheme R_i. If the fact that w_1 has a distinguished variable for attribute A_{i_j} implies that w_2 has the same variable, then w_2 subsumes (or covers) w_1.

Example: In the tableau below, the second row subsumes the first row.

A_1	A_2	A_3
a_1	b_1	b_2
a_1	b_2	a_3

2.3. Data dependencies

The integrity constraints most frequently mentioned in this thesis are *functional dependencies* and *join dependencies*. A set of tuples {t} (relation) *satisfies* the functional dependency X→A if

$$\forall\ t_1,t_2 \in \{t\}\ \Pi_X t_1 = \Pi_X t_2 \Rightarrow \Pi_A t_1 = \Pi_A t_2.$$

A relation r *satisfies* the join dependency $*[R_1 \cdots R_n]$ if

$$r = \Pi_{R_1}(r) \times \cdots \times \Pi_{R_n}(r).$$

Given a set of functional dependencies F, there are inference axioms that can be used to infer all functional dependencies implied by F. The *closure* of F, denoted F^+, is the smallest set containing F such that these axioms cannot be applied to yield any functional dependency not in the set. If a relation scheme is subject to a set of functional dependencies, this set will be assumed to be a nonredundant set, where all dependencies are full. A functional dependency X→Y $\in F^+$ is *full* if $\forall\ X' \subseteq X,\ X' \rightarrow Y \in F^+ \Rightarrow X'=X$. A set of functional dependencies F is *nonredundant* if F does not contain a proper subset F′, such that F≡F′.

Relation schemes are *normalized* — i.e., decomposed by special projections — to eliminate some types of redundancy and solve some update problems. The only normal form referred to in the thesis is *Boyce-Codd Normal Form (BCNF)*. A relation scheme R_i is in BCNF if

(R_i is subject to X→A and A $\not\subset$ X) ⇒ X→R_i,

i.e., all left hand sides of non-trivial dependencies determine the whole scheme.

Template dependencies [SAD80, SAD82] generalize join dependencies, and are represented in a tableau-like format, being composed of two parts: a set of *hypothesis rows*, followed by one *conclusion row*. The conclusion row is separated from the hypotheses by a horizontal line. Each row consists of abstract symbols representing attributes. If at most one symbol is repeated in each column, each such symbol is considered by some authors to be a distinguished variable (e.g., [AHO79]); others assume that there is no distinction between types of variables (e.g., [SAD80]). For a relation to satisfy a template dependency, whenever there is a valuation that maps hypothesis rows to tuples in the relation, the valuation must also map the conclusion row to a tuple in the relation. Template dependencies can be considered as similar to logic clauses, being interpreted as "if a relation contains a set of tuples that correspond to a valuation of the hypothesis rows, then the relation is consistent only if it contains a tuple corresponding to the same valuation applied to the conclusion row".

Template dependencies can be *typed* (in which case a symbol cannot appear in more than one column) or *untyped* (when a symbol can appear in several columns). *Full* template dependencies occur when every variable in the conclusion row also appears in some hypothesis row; *partial* dependencies are characterized by the fact that the conclusion row contains symbols that do not appear in any hypothesis row. Partial template dependencies correspond to embedded constraints. If the dependency corresponds to an embedded constraint over the set of attributes X it is also called *X-partial.*

Example: Let X=(ABC), and consider the X-partial dependency that follows.

$$\begin{array}{cccc}
A & B & C & D \\
\hline\hline
a_1 & b_1 & c_2 & d_1 \\
a_1 & b_2 & c_1 & d_2 \\
\hline
a_1 & b_2 & c_2 & d_3
\end{array}$$

This template corresponds to an embedded multivalued dependency over attributes (ABCD), denoted $A \rightarrow\rightarrow B \mid C$. Template dependencies will be denoted by $(w_1 \cdots w_s) | w_{s+1}$, where the bar separates the conclusion row from the hypotheses.

A template is called *simple* if each column has at most one repeated variable [AHO79].

Example: The template dependency to the left is simple; the one on the right is not.

$$\begin{array}{ccc}
A & B & C \\
\hline\hline
a_1 & b_1 & c_1 \\
a_1 & b_2 & c_2 \\
a_2 & b_3 & c_1 \\
\hline
a_1 & b_2 & c_1
\end{array}
\qquad
\begin{array}{ccc}
A & B & C \\
\hline\hline
a_1 & b_1 & c_1 \\
a_1 & b_2 & c_2 \\
a_2 & b_3 & c_1 \\
\hline
a_2 & b_2 & c_1
\end{array}$$

Sagiv [SAG85] has proved that simple full template dependencies correspond to join dependencies.

Sadri and Ullman [SAD80, SAD82] derived a set of inference rules for template dependencies which is complete and sound for infinite relations. Of these, the following rules will be used in the thesis:

a) The trivial template dependency

The dependency $w_r \mid w_r$ is true for any row w_r.

b) Renaming (or substitution)

Let σ be a mapping that maps a template row $(a_1 \cdots a_k)$ to $(b_1 \cdots b_k)$. The template dependency $(w_1 \cdots w_r) \mid w_{r+1}$ implies $(\sigma(w_1) \cdots \sigma(w_r)) \mid \sigma(w_{r+1})$ if

(1) given attributes A and B such that A$\neq$B, then $\sigma(w_i(A)) \neq \sigma(w_i(B))$, i.e., σ cannot map two different symbols to the same symbol;

(2) σ does not identify a symbol that appears only in the conclusion row with any other symbol.

Example: In the partial template dependency shown previously, if the first row is mapped into the row (abcd), the template after substitution rules are applied becomes

A	B	C	D
a	b	c	d
a	b_2	c_1	d_2
a	b_2	c	d_3

A tableau T1 contains a tableau T2 over the set of instances to which they apply (T2 $\subseteq$ T1) if, for every instance i in the set, $T2(i) \subseteq T1(i)$. A homomorphism h from T1 into T2 is a mapping - called a *containment mapping* - of the variables of T1 into those of T2 that preserves the matching of distinguished variables that exists among the template rows. If there is a homomorphism from $T1$ into $T2$, then $T2 \subseteq T1$ [AHO79]. Template rows can be eliminated whenever there is a containment mapping from a template dependency into itself whose image does not include all the hypothesis rows of the dependency. A template dependency is said to be *minimal* if no row can be eliminated by some containment mapping.

Example: The template dependency on the left can be reduced to the one on the right, which is minimal (and represents the dependency A$\twoheadrightarrow$B|C).

A	B	C
a_1	b_1	c_1
a_1	b_2	c_2
a_1	b_3	c_3
a_1	b_1	c_3

A	B	C
a_1	b_1	c_1
a_1	b_3	c_3
a_1	b_1	c_3

2.4. Tableau chases

Several properties of relation schemes are derived by executing a process called *chase*. Chasing a tableau means applying a set of rules that simulate enforcement of functional dependencies (F-rules) and join dependencies (J-rules) over the relation scheme which is represented by the tableau. The chase generates a sequence of tableaux $T_1 \cdots T_j$, which ends when no more tableaux are generated (i.e., $\nexists$ k,j | $T_k \neq T_j \wedge$ k>j). Chase rules are defined as follows:

a) F-rule

F-rules are applied for every functional dependency X→A. Let tableau T have rows w_1 and w_2 such that $w_1(X)= w_2(X)$, $w_1(A)=\xi_1$, and $w_2(A)=\xi_2$, and suppose $\xi_1 \neq \xi_2$. The F-rule is applied by unifying the variables ξ_1 and ξ_2, which creates new rows and forms a new tableau. This unification is done by choosing one of the variables as the final result. If either ξ_1 or ξ_2 is a distinguished variable, then the result is that distinguished variable (since only one distinguished variable is allowed per attribute); otherwise, the unification chooses the non-distinguished variable with the smaller subscript.

Example: Application of the F-rule $A_1 \rightarrow A_2$ to the tableau on the left results in the tableau to the right:

A_1	A_2	A_3
a_1	b_1	b_2
b_3	b_1	a_3
a_1	a_2	b_4

A_1	A_2	A_3
a_1	a_2	b_2
b_3	a_2	a_3
a_1	a_2	b_4

J-rules are not used in this thesis, and their description can be found in Maier's book [MAI83, pp. 163]. This chapter describes instead T-rules, which are similar to J-rules and refer to chasing with template dependencies.

b) T-rule

Let TD=$(w_1 \cdots w_s)|w_{s+1}$ be a full template dependency on scheme R, and T_1 be a tableau on R. If there exists a valuation ρ such that $\rho(w_1 \cdots w_s) \subseteq T_1$ but $\rho(w_{s+1})$ is not in T_1, add $\rho(w_{s+1})$ to T_1.

Example: Let TD be the template dependency on the left, and T_1 the tableau on the right. Chasing T_1 with TD generates the rows (abe) and (adc).

$$\begin{array}{ccc} A & B & C \\ \hline\hline a_1 & b_1 & c_1 \\ a_1 & b_2 & c_2 \\ \hline a_1 & b_1 & c_2 \end{array}$$

A	B	C
a	b	c
a	d	e

The chase computation with F-rules and J-rules is a finite replacement system with the Church-Rosser property [MAI83]. That is, for an initial tableau and a set of dependencies, F and J-rules can be applied in any order, and the final tableau will always be the same up to renaming. Finite chases do not include embedded join dependencies, since in such a case there would be a need for creating new variables every time a new row is added to the tableau, eliminating the guarantee of a finite chase process. (The proof of finiteness uses the fact that no new symbols are generated). For some cases, it is convenient to consider infinite relations as the result of chasing embedded join dependencies [e.g., MAI84].

2.5. Tableau queries

Since views are usually considered to be single-relation images of queries, their creation can sometimes be simulated by tableau operations. The only queries that can be modelled by tableaux are those involving select on equality, project, and join operators. Select can be extended to inequalities by using additional tablees (see Ullman [ULL82]). In special cases, union and difference operators can be considered as well. Queries by tableaux are used to optimize query evaluation, minimizing the number of joins. Furthermore, they can also be modified to obey some special dependencies.

Tableau queries, which are a modified version of tableaux, can either be processed against a single relation, or against several relations. The latter can only be implemented if all relations are projections of a common instance (which is an underlying assumption in the universal relation model). Besides having distinguished and non-distinguished variables, tableau queries also contain constants and blanks. Furthermore, each tableau is preceded by a special row — the *summary row* — which describes the general format of the tuples answering the query. Variables in a tableau query cannot appear in more than one column, and only summary rows can contain blanks. Any distinguished variable that appears in a column must also appear in the summary row, which cannot contain non-distinguished variables.

Example: For R_1=(AB), R_2=(BC), the query $\Pi_{AC}R_1 \times R_2$ is represented by the tableau

A	B	C
a_1		a_3
a_1	b_1	b_2
b_3	b_1	a_3

The first row is the summary row, the second and third rows stand for tuples in R_1 and R_2, respectively. The query $\sigma_{B=0}R_1 \times R_2$ is represented by

A	B	C
a_1	0	a_3
a_1	0	b_2
b_3	0	a_3

Tableau queries can be chased in a manner similar to traditional tableaux, and a variant of the row subsumption concept is used to derive some of their properties. Tableau queries can be modified so that they are no longer based on the assumption that all relations used to form the query are projections of a common instance. In this case, each row is tagged with indication of the relation scheme it represents, and the row contains blanks for attributes not in the scheme.

Example: Let R={R_1<ABC,{A→B}>, R_2<ABD,{A→B}>}. The query $\Pi_{CD}R_1 \times R_2$ is translated into the tagged tableau

A	B	C	D	(Tag)
		a_3	a_4	
b_1	b_2	a_3		(ABC)
b_1	b_2		a_4	(ABD)

whereas Π_{CD} ($\Pi_{AC}R_1 \times \Pi_{AD}R_2$) is translated into

A	B	C	D	(Tag)
		a_3	a_4	
b_1	b_2	a_3		(ABC)
b_1	b_3		a_4	(ABD)

These queries do not necessarily result in the same set of tuples, since there is no condition that the dependency A→B for R_1 must denote the same function as the dependency A→B for R_2. For instance, for the extensions r_1={(1,2,5) (2,3,4)} and r_2={(1,3,7) (2,3,8)} (which cannot be processed by non-tagged tableau queries) the first query would result in the tuple (4,8), whereas the second would result in the set {(5,7) (4,8)}.

If the last tableau were instead a non-tagged query, it could be chased with F-rules, and the chase would yield $b_3 = b_2$. For tagged tableau queries, chase rules — usually F-rules only — can only be applied over rows with the same tag (i.e., referring to the same relation scheme).

The chase computation can also include rules for template dependencies (T-rules), but in this case it may not be finite [FAG83a]. In some special cases, the implication problem for template dependencies results in a finite chase process. The chase is finite, for instance, if the set of constraints C to apply consists of a single dependency, or if the dependencies involved are full. A set of template dependencies cannot be always combined into a single template (unless they are all full [FAG83a]). In fact, the conjunction of a finite set of template dependencies (not necessarily full) may not be equivalent to any single template dependency.

2.6. Basic notation and assumptions

In this thesis, a relation scheme will always be considered together with the set of constraints that apply to the scheme, and does not necessarily obey the universal model. Inter-relation constraints will not be considered. Therefore, unless otherwise specified, "relation scheme" and "database scheme" will in reality refer to sets of attributes and constraints, denoted R=$\{R_i <\{A_{i_j}\},\{C_{i_n}\}>\}$, where $\{A_{i_j}\}$ represent the attributes of R_i, and $\{C_{i_n}\}$ the constraints of R_i. All attribute domains will contain at least two elements — empty domains have no meaning and a singleton domain for some attribute A corresponds to imposing the functional dependency (U)→A on the database scheme, where (U) represents the set of attributes of the universe. A view is defined as

$$V = (R, q_v)$$

where q_v is the view generating function which, applied to R, will result in the desired query image. The generating functions analyzed will be assumed to be monotonic (i.e., if r $\subseteq$ s then q_v(r) $\subseteq$ q_v(s)) and formed by selection (σ), projection (Π) and natural join ($\times$) operators. Any view that satisfies these requirements will be referred to as an *SPJ view*.

The template dependencies considered in this thesis will be all those that belong to the class of embedded join dependencies (e.g., multivalued dependencies). Such templates may be manipulated as simple, typed and minimal tableaux (each of which represents some join dependency).

Chapter 3

The update validation algorithm

This chapter describes the *update validation algorithm*, a general algorithm to validate individual update policies at design time. The algorithm indicates whether there can be any side effects to a policy, allowing the view designer to determine when a desired update will, in fact, occur in the view. It is assumed that view access mechanisms (such as authorization and serialization of operations) are handled elsewhere by the system, and view interference will not be considered.

This chapter consists of two parts. The first contains the basic framework and definitions used throughout the thesis. The second part analyzes the proposed algorithm in a top-down fashion, and consists of a general overview followed by a detailed description of each of its stages. An example at the end of the chapter illustrates the execution of the algorithm.

3.1. Basic framework

A view V=(R,q_v) will be defined over a relational database scheme R=$\{R_i <\{A_{i_j}\},\{C_{i_n}\}>\}$, where $\{C_{i_n}\}$ is assumed to be composed of a set of full functional dependencies and at most one minimal typed simple template dependency. There will be no further integrity constraints imposed by the view; the only constraints allowed in this model are those referring to the underlying relations.

To avoid the complexity inherent in inference problems for template dependencies [FAG83a, see also Chapter 2], it is assumed that templates cannot span relation schemes, and that there is at most one template dependency for any underlying relation scheme (although there is no limitation on the number of functional dependencies allowed). Inter-relation constraints will not be considered. Thus, the schemes analyzed are independent in the sense that local (relation) consistency implies global (database) consistency.

3.1.1. Partitionable BCNF

Chapter 6 shows that there are cases in which functional dependencies interact in such a way that certain updates may not be performed. The concept of BCNF partitions, used throughout the thesis, is needed to prevent this type of situation.

$R_i <\{A_{i_j}\},\{C_{i_n}\}>$ (as restricted above) is said to be in *Partitionable BCNF* if there exist sets of attributes $\{P_i\}$ such that

$$\{A_{i_j}\} = \{P_1\} \cup \{P_2\} \cdots \cup \{P_l\}$$

where

$$\forall \text{ k,j } \{P_k\} \cap \{P_j\} = \emptyset \text{ and}$$

$$\forall X_k, A_k \ [\ (X_k \rightarrow A_k) \in \{C_{i_n}\}] \Rightarrow [\exists \text{ j} \mid X_k A_k \subseteq \{P_j\} \wedge X_k \rightarrow \{P_j\}].$$

In words, R_i is in Partitionable BCNF if its attributes can be partitioned into BCNF *slices* (or *partitions*), where the set of functional dependencies given for the attributes of each slice is local to that slice (no functional dependency connects any two slices). A partition will be said to be *trivial* if no functional dependencies apply to its attributes.

Example: Consider R_1<ABCDEFG, {A→B,C→DE,E→C, TD}>, where TD is a simple minimal template dependency, whose interaction with the other constraints in this scheme is not taken into consideration for determining the partitions. R_1 is in Partitionable BCNF, with partitions {(AB), (CDE), (FG)}. Note that {FG} is a trivial partition. As a second example, R_2<ABCD, {A→BCD,B→A}> is in BCNF and thus in Partitionable BCNF. On the other hand, R_3<ABCDE, {A→B,B→C}> is not in Partitionable BCNF: even though (DE) is a trivial partition, (ABC) cannot be partitioned into BCNF slices that do not interact.

3.1.2. Update translations

Let V=({$R_1 \cdots Rn$}, q_v) be a view and U_v an update requested through this view. The *complete update translation* or *complete policy* $u_v = \{u_i\}_{i=1..n}$ is the translation of U_v into a set of *underlying* or *isolated update operations* u_i on relations r_i. Let $u_i(r_i)$ represent the updated relation r_i and Δ_i is the set of modifications in r_i due to update u_i. The updated view can be computed as $q_v(u_1(r_1) \cdots u_n(r_n))$. The *isolated contribution* to the view of updating r_i is computed as $q_v(r_1 \cdots \Delta_i \cdots r_n)$.

3.1.3. Forced and conditional updates

This thesis allows each isolated update in a complete policy to be executed according to a pre-defined action: force or conditional. *Forcing* an update means that the update must be performed, and that *additional* updates may have to be executed as well, in order to achieve a consistent final state. *Conditional* updates are executed only if no additional updates to the underlying database are needed, and correspond to the traditional approach to updating relations. Forcing updates rather than forbidding them follows the suggestion of Fagin, Vardi and Ullman [FAG83]. However, unlike Fagin *et al.*, who also accept changing integrity constraints, the present work restricts additional updates to the extensional level, and the integrity constraints are assumed fixed.

Example: If R_1=<(Name,Dept), {Name→Dept}> contains the tuple (Ethel,CS), a request for conditional insertion of (Ethel,Math) is rejected, since it cannot be performed without deleting the tuple (Ethel,CS). A request for *forced* insertion of the same tuple, however, can be accepted (under the presumption 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 (Ethel,Math).

3.1.4. Symbolic tuple expressions

Instead of actual tuples, the update validation algorithm manipulates *symbolic tuple expressions* over variables λ_i, where each variable is designated as either *parametric* or *placeholder*. Let T be a set of tuples defined over a set of attributes $\{A_i\}$. A symbolic tuple expression {t} for $\{A_i\}$ is an ordered string of variables $\{\lambda_1 \cdots \lambda_n\}$ such that each λ_i represents an attribute A_i. A *valid valuation* ϕ of {t}, denoted ϕ({t}), is a mapping that transforms each symbol λ_i into a value $\phi(\lambda_i) \in \text{dom}(A_i)$. {t} *describes a tuple* $t_1 \in$ T if there exists a valid valuation ϕ({t}) such that $\Pi_{\{A_i\}}\phi(\{t\}) = \Pi_{\{A_i\}}t_1$. {t} *describes a set of tuples* T if there exists a set of valid valuations of {t} such that each tuple in T can be mapped into {t} by a valuation in this set and the projection of the set of valuations on the attributes represented by the parametric variables results in a single tuple. In particular, the symbol $\{t\}_i$ denotes a symbolic tuple expression that describes a set of tuples to be updated for scheme R_i.

Example: Let (AbC) be a symbolic tuple expression, where the capital letters stand for parametric variables and each attribute is defined over a set of digits. (AbC)={(123), (143)} is a set of valid valuations, since all tuples have the same value for A and C. However, (AbC) does not describe {(123), (148)} because there are two different valuations for parametric symbol C.

Symbolic tuple expressions are manipulated by the update validation algorithm as if they were real tuples. The algorithm determines, at design time, the description of a set of tuples which result from an update; at execution time, valid valuations of these expressions yield actual tuples. Note that *execution time*, or *run time*, in this thesis, refers to the stage in the database lifecycle where the view has already been designed, and is being actually queried/updated by the user, as opposed to the *design time* when the update validation algorithm is executed.

As noted earlier, each expression involves *parametric* and *placeholder* variables. Parametric variables (denoted by uppercase characters) represent specific attribute values at execution time, and placeholder variables (in lowercase characters) represent all other attributes. Constants are a special case of parametric variables.

Example: Let R_1=(Parent, Child, Dept) have an instance {(John,Mark,CS), (John,Linda,CS), (Mary,Paul,Math)}. (John,*,CS) refers to the first two tuples of this instance. The corresponding symbolic expression is $\{t\}_1$=(PcD), i.e., the set of tuples with specific values for attributes Parent and Department to be given at execution time. Notice that this expression may also describe the tuple (George,*,Biology), etc. Update requests can thus be specified in terms of symbolic expressions. For example, the request to delete (John, *, CS) is an instance of deletion via the tuple expression (PcD), which includes the deletion of tuples of type $\exists$ x | (John, x, CS) $\in r_1$, when John and CS are substituted for the parametric variables P and D.

Parametric variables can represent two types of values: user-entered or system-generated; all valuations of placeholder variables are system-generated. User-entered parametric variables correspond to attributes whose specific value (valuation) will be provided by the user at execution time. System-generated variables correspond to attributes that have their value determined by the system at execution time. All system-generated variables have indices, which originate directly from the expressions manipulated by the algorithm. The effect of assigning a parametric value to a variable is that of restricting the range of values the corresponding attribute may have. When the value of an attribute changes, the (parametric) variable representing the old value is marked, to differentiate it from the new value. Different parametric symbols stand for separate attribute values at execution time.

The symbolic expressions that may appear in the algorithm's output contain, therefore, for an attribute A

a_i - any attribute value

A - a specific user-entered value

A' - a specific value denoting the old value of A, different from the new value A.

A_i - a specific system-generated value, determined by a functional dependency, or a range of values, determined by a select operator.

A'_i - a specific value (system-generated) denoting the old value of A_i, different from the new value A_i.

$A \rightarrow B_i$ - the specific value B_i is determined by the specific value of A, for a functional dependency A→B .

Input expressions that consist exclusively of parametric variables refer to a single tuple, e.g., $\{t\}_i$=(ABC) refers to a tuple with specific values for attributes A, B and C at execution time. For the purposes of the update validation algorithm, insertions can only be specified one tuple at a time (i.e., through parametric symbolic expressions), which is similar to the approach of Furtado *et al.* [FUR79]. Input expressions consisting of both parametric and placeholder variables can be accepted for deletion or change operations, where the set of values to be assigned to the placeholder variables depends on tuples that already exist in the database. A request for deleting a tuple (AbCD) from view V, for instance, is interpreted as a request for deleting all tuples of the form $\exists$ b| (AbCD) $\in$ V.

In this thesis, the sentence "the set of tuples described by an expression {t} is deleted(inserted)" is often shortened to "{t} is deleted(inserted)", when the context is clear.

3.1.5. Matching and equivalence

Let a template dependency be defined as $w_1 \cdots w_s \mid w_{s+1}$. Row w_j *matches* row w_k *over* a set of attributes $\{a_M\}$ if the template contains the same symbols for these attributes in both w_j and w_k. In other words, w_j matches w_k over $\{a_M\}$ if any valuation will assign identical values to the attributes $\{a_M\}$ in w_j and w_k. Similarly, given a set of attributes Y, two symbolic expressions *match* over Y if both have the same symbols for Y, and two tuples t_1 and t_2 *match over* Y if $\Pi_Y t_1 = \Pi_Y t_2$.

Consider the functional dependency X→A, and let {t} and {w} be symbolic tuple expressions defined over the same set of attributes, describing sets of tuples T and W, respectively. {t} and {w} *agree over XA* if (T∪W) satisfies X→A. {t} and {w} *disagree over XA* if (T ∪ W) violates X→A. {t} *is equivalent to* {w} ({t} $\equiv$ {w}) whenever both describe exactly the same set of tuples. In other words, the set of valid valuations for {t} is exactly the set of valid valuations for {w}. {t} is *subsumed by* {w} ({t} $\subseteq$ {w}) whenever the set of tuples described by {t} is a subset of the tuples described by {w}.

Example: Consider analyzing the equivalence of the expressions $(a_1 B_1 c_1)$ and $(a_2 B_2 c_3)$. Assume all system-generated placeholder variables, for each attribute, range over the same set of values. If the set of valid valuations for these expressions determines that B is such that $B_1 > 10$ and $B_2 > 10$, these expressions are equivalent. However, if $B_1>10$ and $B_2<10$ they are no longer equivalent. Finally, if $B_1 \geq 10$ and $B_2>10$, then the first expression subsumes the second.

3.1.6. Symbolic valuation of a template dependency

A template dependency $w_1 \cdots w_s \mid w_{s+1}$ over scheme R_i can be seen as a logic sentence $w_1 \wedge \cdots \wedge w_s \Rightarrow w_{s+1}$. In the context of deductive logic databases, similar structures are treated as recursive axioms (e.g., Minker and Nicolas [MIN83]). A *symbolic valuation* of a template dependency occurs when a symbolic tuple expression (or set of expressions) replaces a row (or set of rows) of the template, and substitution inference rules are applied. The result is denoted $(\overline{w}_1 \cdots \overline{w}_s) | (\overline{w}_{s+1})$, where each $\overline{w}_j$ is a symbolic tuple expression. This can also be seen as a logic sentence $\overline{w}_1 \wedge \cdots \wedge \overline{w}_s \Rightarrow \overline{w}_{s+1}$, which is true if there exists a valid valuation ϕ such that $\bigwedge_j [\phi(\overline{w}_j) \in r_i]$. The term $(\overline{w}_j \in r_i)$, often used by the validation algorithm, represents a condition which is true if there exists a valid valuation ϕ such that $\phi(\overline{w}_j) \in r_i$.

3.2. Complete example

The update validation algorithm processes symbolic tuple expressions. In order to help understand the sections that follow, this section simulates the execution of the algorithm using actual instances instead of symbols. Consider the attributes C=Course, M=Meet_time, S=Student, P=Prof. Let R= $\{R_1<$(C,M,S),{C→→M|S, (MS)→C}$>$, $R_2<$(C,P), {C→P}$>$}, and a view defined by $q_v = R_1 \times R_2$. Consider the instances

C	M	S
Phys	Mon 10	James
Phys	Mon 10	Paul
Phys	Wed 10	James
Phys	Wed 10	Paul
Biol	Mon 10	Mark
Biol	Mon 10	John
Biol	Tue 18	Mark
Biol	Tue 18	John
Chem	Fri 15	Mark

C	P
Phys	Prof1
Biol	Prof2

Suppose the tuple (Chem, Mon 10, James, Prof2) is to be inserted into the view, and the associated complete translation is u_v= { u_1 = Force insertion of (Chem, Mon 10, James) into R_1; u_2 = Insert (Chem, Prof2) into R_2 conditionally}. After syntax verification, the first action taken by the algorithm is to check what modifications Δ_i each relation must undergo in order to support the isolated updates requested.

Computing Δ_1 for relation r_1

Forcing indicates that the insertion must be performed, even if it means changing information already in the database.

: (Chem, Mon 10, James) can only be inserted if t_1 = (Phys, Mon 10, James) is deleted, because MS→C.

: Since (Chem, Fri 15, Mark) is in r_1, and C→→M|S, the requested insertion also

requires insertion of (Chem, Fri 15, James) and (Chem, Mon 10, Mark).
: However, this last insertion can only be performed if t_2 = (Biol, Mon 10, Mark) is deleted, again to satisfy MS→C.
: Finally, deletion of t_1 and t_2 cannot be accomplished because there are other tuples with C=Phys and C=Biol that require the existence of t_1 and t_2, to satisfy C→→M|S. Deletion of t_1 can be achieved if either (Phys, Wed 10, James) or (Phys, Mon 10, Paul) is also deleted; deletion of t_2 can be achieved if either (Biol, Mon 10, John) or (Biol, Tue 18, Mark) is also deleted. Suppose that some pre-defined translation rule (see Section 3.4.1.c) determines that (Phys, Wed 10, James) and (Biol, Mon 10, John) should be chosen for deletion.

The final set Δ_1 of insertions and deletions is thus

insertions:
  (Chem, Mon 10, James) - requested
  (Chem, Mon 10, Mark) (JD)
  (Chem, Fri 15, James) (JD)
deletions:
  [(Phys, Mon 10, James)] (FD)
  [(Phys, Wed 10, James)] (JD)
  [(Biol, Mon 10, Mark)] (FD)
  [(Biol, Mon 10, John)] (JD)

where (FD) and (JD) indicate whether the functional dependency or the join dependency, respectively, caused the tuple to be inserted (deleted). These additional updates occur only because of the relation's initial state. If, for instance, Mark did not take Biology at 10 on Mondays, there would be no need to delete (Biol, Mon 10, John). For this reason, each underlying update is associated with the initial state under which it will occur (called *relation state description*).

Computing Δ_2 for relation r_2

Conditional insertion of (Chem, Prof2) can be performed since it does not violate C→P. Had there already been some other professor in Chem, however, this update would not have been allowed.

At this point, the modifications Δ_1 and Δ_2 have been computed, and stored in a table called *underlying modification table.* The next part of the algorithm computes the changes that happen to the view given these changes in the underlying relations. In fact, the changes due to each relation are processed separately as if the other relations had not changed — i.e., the isolated contribution of each relation to the view is computed. This is done in a way similar to query computation.

Contribution of Δ_1

The view will contain the new tuples
  (Chem, Mon 10, James, P_1)
  (Chem, Mon 10, Mark, P_1)
  (Chem, Fri 15, James, P_1)
while the tuples
  (Phys, Mon 10, James, P_2)
  (Phys, Wed 10, James, P_2)
  (Biol, Mon 10, Mark, P_3)

(Biol, Mon 10, John, P_3)
are deleted. P_1, P_2, P_3 are system-generated parametric variables which will be assigned values from relation r_2, i.e., $P_1 = P_3$ = Prof2 and P_2 = Prof1. These insertions and deletions result from the original insertions and deletions in Δ_1 only because tuples {(Chem, Prof2), (Phys, Prof1), (Biol, Prof2)} exist in r_2. Thus, the changes in the view are *conditional* to some database state. Such a set of conditions associated with each view update is called the *database state description*, which is computed and stored in the *database state table.*

Contribution of Δ_2

The view will contain new tuples {t}, where Π_{CP} {t} = (Chem, Prof2), and the values for attributes M and S are taken from tuples in r_1 that contain C=Chem, and *which are not deleted* in Δ_1, since deleted tuples cannot contribute to view insertions. Thus, besides the insertion of the tuples already mentioned, Δ_2 additionally causes (Chem, Fri 15, Mark, Prof2) to appear in the view.

Set of possible updates

The set of all possible updates that can occur to the view given the complete policy specified is thus given by combining the set of isolated contributions of each relation to the view, i.e.

Contribution of both Δ_1 and Δ_2

Insertion of (Chem, Mon 10, James, Prof2), (Chem, Mon 10, Mark, Prof2), and (Chem, Fri 15, James, Prof2).

Contribution of Δ_2 only

Insertion of (Chem, Fri 15, Mark, Prof2)

Contribution of Δ_1 only

Deletion of (Phys, Mon 10, James, P_2), (Phys, Wed 10, James, P_2), (Biol, Mon 10, Mark, P_3) and (Biol, Mon 10, John, P_3)

Reviewing what has been shown in this example, the algorithm is processed along the following lines

1) Validate input

2) Determine changes Δ_i to each relation, given the translation proposed, and the relation state for which each change occurs. Store changes in the underlying modification table.

3) Produce as output the isolated contribution of each Δ_i to the view, and the database state description for which the contribution may occur.

A few other keypoints should be mentioned. Notice that an underlying forced insertion (e.g., in r_1) may cause tuple deletions from the view. Furthermore, isolated contributions, alone and in combinations, yield all side effects. By analogy, given the algorithm's symbolic output, the set of valid valuations of the (symbolic) isolated contributions yield the set of all possible side effects that may occur in response to a given translation (see proof in Chapter 4). Finally, view updates are conditional on the initial relation's states. The sections that follow formalize these notions.

3.3. The update validation algorithm - functional description

Whereas most authors consider requests for tuple replacement to be equivalent to a transaction of type <single deletion, single insertion>, here replacements will be treated as separate atomic operations, here called *changes*. The algorithm can thus be used to validate insertions, deletions, and requests for tuple replacements.

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 insertions of tuples into the underlying relations, or replacements of null by non-null values in some underlying tuples [LIN78, DAY82]. Following again the thesis' basic premise that the set of admissible updates should be as little restricted as possible, the focus of concern will be that of composition of underlying updates, but no restriction is imposed on the types of operation requested. Thus, a given view update can be translated into a series of forced and conditional changes, insertions and deletions, and it is possible to effect a deletion or a replacement by an underlying set of insertions, as was shown in Section 3.2.

The update validation algorithm receives as input the database scheme, the view generating function, the desired view update and the complete update translation for this update. The output predicts the effects the translation has on the view, as a function of the initial database state. As such, it can be seen as a transformation that takes the desired view update and a complete policy into the set of all possible effects this policy may have on the view. Both input and output are specified in terms of symbolic tuple expressions.

The input to the algorithm is a pair $<\mathrm{R},q_v>$, representing view formation and consistency information; and an update rule to be validated. A syntactic restriction imposed on the database scheme is that each of its component relations should be presented in Partitionable BCNF. Each update rule can be considered to be a procedure of the form <*view update desired,{isolated operations}*>, which is activated at run time by a specific update request. *View update desired* describes the update operation to be performed, and plays the role of procedure header. The set *isolated operations* is the executable body, and forms a complete policy.

Each *isolated operation* is specified as the 4-tuple

<*Op, tuples affected, underlying relation* R_i, *exception action*>.

Op indicates the type of operation, which can be *In* (insertion),*De* (deletion) or *Ch* (change);

tuples affected is $\{t\}_i$ or $\{t,t'\}_i$
(the latter format is used when the operation is a change: $\{t\}_i$ replaces $\{t'\}_i$);

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.

The algorithm first checks the input to see if it is syntactically correct (see details in subsequent sections). Next, it computes what are the updates that must be performed at each scheme R_i (e.g., Δ_1 and Δ_2 in Section 3.2). These updates are denoted Δ_i, and consist of a set of symbolic tuple expressions. These expressions describe tuples to be inserted or deleted in each relation in order that the update requested at the translation, $u_i=(Op_i,\{t\}_i,R_i,exception)$, be performed, and each final relation state satisfies $\{C_{i_n}\}$. The *underlying modification table* is a table that contains all sets Δ_i.

The execution of the (set of) updates described by each expression in Δ_i is conditional on the relation state before the update — the *relation state description*. This state is described in terms of existence of symbolic tuple expressions, and the rules for determining it are denoted $State(Op_i)$.

Next, the isolated contribution to the view of the updates described in Δ_i is computed, for each scheme R_i. This is achieved by means of applying the view generating function q_v to the expressions in the underlying modification table, creating for each scheme R_i a *tableau sequence* $T_{1_i} \cdots T_{k_i}$. Tableau T_{1_i} contains the expressions in Δ_i, as well as one symbolic expression for each relation scheme $R_{j \neq i}$. Each tableau T_{j_i} in the sequence is obtained by applying one operator $(\times, \Pi, \sigma)$ of q_v to the rows of $T_{(j-1)_i}$ and checking functional dependencies. Each row of tableau T_{k_i} consists of a symbolic tuple expression that describes a set of view updates (tuples) that may occur as a result of the underlying updates to R_i described in T_{1_i}. In other words, the rows of T_{k_i} represent the isolated contribution of R_i to the view.

Each tableau sequence is associated with a *database state table*. Initially, this table contains the relation state description (for R_i) determined according to rules $State(Op_i)$. This table is modified during the creation of the associated tableau sequence, in order to reflect the conditions under which each transition from $T_{(j-1)_i}$ to T_{j_i} can occur. Each row of the final database state table is associated with one row W of T_{k_i} and contains a conjunction of variable valuations and symbolic expressions. These expressions describe a class of initial database states for which the set of view updates described in W will appear in the view. Each database state in the class corresponds to a valid valuation of W. Each row of the table is said to contain a *database state description*.

Example: Let $q_v = \sigma_{A>10}(R_1 \times R_2)$, R=$\{R_1$<AB,{}>, R_2<BC,{B→C}>$\}$. For Δ_1='ab', and Δ_2='Bc', the result of the update validation algorithm will be seen to be $\{t\}_{12}=A_1BC_1$, which limits the set of valid valuations for view tuples to those with specific (sets of) values for A, B and C. Besides indicating what are the conditions for generating Δ_1 and Δ_2, the database state table entry for $\{t\}_{12}$ records:
[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 limits the range of the corresponding attribute.
Both A and C are assigned indices to denote that the corresponding values are not directly determined by the user.

Thus, given the complete policy $u_v=\{u_i=(Op_i,\{t\}_i,R_i,exception)\}_{i=1..n}$, the complete output is the set of rows in

$\{T_{k_i}$ ∪ database state table for sequence i $\}_{i=1..n}$.

Therefore, the update validation algorithm is a function that transforms an incompletely specified view update and its translation u_v into the precise functional definition of this translation in terms of its effects in the view.

The next section shows how underlying modification and database state tables are built, and describes the rules for generating a tableau sequence, denoted $Ex(q_v)$. Since tableau sequences are independent of each other and processed separately, any sequence will be from now on denoted $T_1 \cdots T_k$. The reader should keep in mind, however, that there are n such sequences, one for each of the relation schemes updated in the complete policy.

3.4. Rules for executing the algorithm

3.4.1. Generation of the underlying modification table

Each underlying update $u_i = (Op_i, \{t\}_i, R_i, exception)$ is processed so as to derive the set of expressions in Δ_i, stored in the underlying modification table, that describe all inserted, changed and deleted tuples in the underlying relation r_i. The set of rules that defines how each Δ_i is generated is denoted $Op(\{t\}_i, R_i, exception)$, where Op is In, De or Ch. Nulls will be interpreted as indicating the existence of an attribute whose value is not known, and will be treated as any other variable. Underlying operations of changing null to non-null values are therefore a special case of changing non-nulls to other non-null values.

The following notation will be used in the remainder of the thesis. The expression [t] denotes that the tuples described by {t} are being deleted from the relation. Pairs of expressions of the form (t,[t']) denote that the tuples described by {t} replace those described by {t'} in the underlying relation. The symbol w_i denotes the i-th row of a template, and $\overline{w}_i$ denotes a symbolic valuation of the same row. When $\{t\}_i$ replaces any template row w_j, all placeholder variables in $\{t\}_i$ are assigned the indices of the corresponding variables in w_j (e.g., if $\{t\}_i$=(AbC) and $w_j=(a_1b_1c_1)$, replacing w_j by $\{t\}_i$ results in $\overline{w}_j=(Ab_1C)$).

References to the updating or the existence of a *template row* will actually refer to the updating or the existence of the *tuples* described by that row. For instance, the statement "(Ab_1C) is deleted if (ABc_2) exists" means that "all tuples in scheme R=(ABC) described by the expression $\exists\ b_1 \mid (Ab_1C) \in r$ are deleted if this relation contains tuples described by $\exists\ c_2 \mid (ABc_2) \in r$ ".

If the exception action is "cond" the update cannot cause any other updates, and thus
$In(\{t\}_i, R_i, cond) : \Delta_i = \{t\}_i$;
$De(\{t\}_i, R_i, cond) : \Delta_i = [\{t\}_i]$;
$Ch(\{t, t'\}_i, R_i, cond) : \Delta_i = (\{t\}_i, [\{t'\}_i])$.
This action is the one assumed by most authors. The rules that follow are for $Op(\{t\}_i, R_i, force)$, where underlying updates other than the requested ones are also generated.

a) 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. Therefore,
$In(\{t\}_i, R_i, force) : \Delta_i = \{t\}_i$;

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

b) R_i is subject to functional dependencies only

Deletion of tuples cannot violate a functional dependency, so $De(\{t\}_i, R_i, force) = [\{t\}_i]$.

For *every* functional dependency X→A that can be violated, both forced insertions and changes may require elimination of some tuples. This is achieved by specifying an expression {w} for attributes in the relation, where {w} consists of placeholder variables only. Based on this expression, a set of change pairs denoted $\{(w_{fd}, [w'_{fd}])\}$ is created such that (w_{fd}) agrees with $\{t\}_i$ over XA, and w'_{fd} describes the tuples that should be deleted:

$\Pi_X w_{fd} = \Pi_X w'_{fd} = \Pi_X \{t\}_i$
$\Pi_Z w_{fd} = \Pi_Z \{w\} = \Pi_Z w'_{fd}$, where Z=($R_i$- (X ∪ A))
$\Pi_A w_{fd} = \Pi_A \{t\}_i$.

For insertions, $\Pi_A w'_{fd} = A'$, where A' is a variable different from A.
For changes, $\Pi_A w'_{fd} = \Pi_A \{t'\}_i$.

Other attributes in w_{fd} may become system-generated parametric variables as a result of checking the dependencies in the partition to which X→A belongs. The set of deletions $\{w'_{fd}\}$ is denoted $\Delta D(t_{FD})$; the set of insertions $\{w_{fd}\}$ is denoted $\Delta I(t_{FD})$.

Thus, $In(\{t\}_i, R_i, force) : \Delta_i = \{(w_{fd}, [w'_{fd}])\} \cup \{t\}_i = \{t\}_i \cup \Delta I(t_{FD}) - \Delta D(t_{FD})$;
$Ch(\{t,t'\}_i, R_i, force) : \Delta_i = \{(w_{fd}, [w'_{fd}])\} \cup (\{t\}_i, [\{t'\}_i]) = (\{t\}_i, [\{t'\}_i]) \cup \Delta I(t_{FD}) - \Delta D(t_{FD})$.

Example: Let R_1<ABCDE,{A→B , C→D}>, which yields the partitions are [AB], [CD] and [E]. Let the expression to process the updates in this relation be {w}=($a_1 b_1 c_1 d_1 e_1$). Forced insertion of (ABCDE) is translated into the set

A	B	C	D	E	[A	B	C	D	E]	
(A	B	c_1	d_1	e_1,	[A	B'	c_1	d_1	e_1])	(fix A→B)
(a_1	b_1	C	D	e_1,	[a_1	b_1	C	D'	e_1])	(fix C→D)
A	B	C	D	E						(insert tuple)

The request (Ch, (ABCDe,A'BCD'e), R_1,force) is translated into

A	B	C	D	E	[A	B	C	D	E]	
(A	B	c_1	d_1	e_1,	[A	B'	c_1	d_1	e_1])	(fix A→B)
(a_1	b_1	C	D	e_1,	[a_1	b_1	C	D'	e_1])	(fix C→D)
(A	B	C	D	e_1,	[A'	B	C	D'	e_1])	(change tuple)

Example: If R_1<ABCD,{AB→CD}>, and the expression used to process updates in R_1 is ($a_1 b_1 c_1 d_1$), then forced insertion of (ABCD) creates the pairs (ABCD_1,[AB$C'd_1$]) and (ABC_1D,[AB$c_1 D'$]).

c) R_i is subject to a template dependency only

If a template dependency $w_1 \cdots w_s \mid w_{s+1}$ is interpreted as an implication clause $w_1 \wedge w_2 \wedge \cdots \wedge w_s \Rightarrow w_{s+1}$, *insertion* of tuples is equivalent to modifying the set of hypotheses $w_1 \cdots w_s$ [NIC78]. Insertion effects are tested separately for each hypothesis row w_j in the template, replacing it by $\{t\}_i$ and applying substitution inference rules to the whole template. Each substituted conclusion row, denoted $(\overline{w}_{s+1})_j$, indicates which additional insertions may be needed to keep the clause valid. The modified hypothesis rows $w_{k,k \neq j}$, constitute the relation state description for which insertion of $(\overline{w}_{s+1})_j$ is needed, if $\{t\}_i$ is inserted. For each substitution, $\{t\}_i$ may play multiple roles (i.e., the state description includes expressions that may correspond to $\{t\}_i$ as well).

X-Partial templates over a scheme R=(XY) represent embedded constraints, and can be seen as composed of an (embedded) template over X plus a set of columns that refer to the attributes (Y) not subject to the constraint. The value of the attributes in Y for each additional insertion $(\overline{w}_{s+1})_j$ cannot be determined from the hypotheses, and must therefore be taken from some designer-defined set of values (e.g., marked nulls). Such attributes are therefore accorded system-generated parametric symbols (uppercase characters with indices).

This process of generating additional insertions is equivalent to computing all symbolic valuations for which $\{t\}_i$ describes tuples in the relation. In this thesis, the process of applying substitution rules starting from a hypothesis row is also referred to as *pushing* $\{t\}_i$ *down the template.* The symbolic valuation that results from replacement of a template row w_k by a symbolic expression {t} will also be referred to as the *mapping generated* by the replacement $w_k \leftarrow \{t\}$.

$In(\{t\}_i, R_i, force) : \Delta_i = \{t\}_i \cup \{(\overline{w}_{s+1})_j\}$, where $\{(\overline{w}_{s+1})_j\}$ is the set of symbolic valuations of conclusion rows obtained in the set of mappings generated by $\{w_j \leftarrow \{t\}_i\}_{j=1..s}$ (describing the tuples that may have to be inserted to maintain the database consistency).

Example: Let R_1<ABCD,{A→→B| C}>, whose ABC-partial template is

A	B	C	D
a_1	b_1	c_2	d_1
a_1	b_2	c_1	d_2
a_1	b_2	c_2	d_3

Forced insertion of (ABCD) results in a set with rows: (ABCD) itself; (Ab_2CD_1) - when (ABCD) replaces the first template row; and (AB$c_2$$D_2$) - when the substitution occurs for the second row. D_1 and D_2 are not subject to any constraint, and thus their values are to be chosen by the system.

Deletion rules are also based on symbolic template valuations. Let again R=(XY) be subject to an X-partial template dependency, and consider the relation after deletion of $\{t\}_i$. If this new relation contains a set of tuples described by $\{t_{rem}\}$ such that $\Pi_X\{t\}_i \equiv \Pi_X\{t_{rem}\}$ (i.e., the relation still satisfies the dependency after the deletion), then deletion of $\{t\}_i$ does not require any further deletions. (Notice that if Y=∅ this can never happen, since deletion of $\{t\}_i$ makes $\{t_{rem}\}$=∅.)

If $\{t_{rem}\}$=∅, then $\{t\}_i$ must be checked against the template dependency. It initially replaces the conclusion row [NIC78], and again substitution inference rules are applied to the rest of the template. This mechanism of applying substitution rules starting from the conclusion will also be referred to as *pushing* $\{t\}_i$ *up the template.* If the expression represented in the conclusion row subsumes some hypothesis row $\overline{w}_j$, then deletion of $\{t\}_i$ also causes deletion of all tuples described by $\overline{w}_j$. If at least one such $\overline{w}_j$ can be found, then $\{t\}_i$ is effectively deleted. If no such row exists, another row may need to be eliminated to invalidate the set of hypotheses. Without loss of generality, this additional row is chosen to be the first substituted hypothesis row that contains at least one parametric variable (or the first row, if no parametric variables appear), and which will be denoted by $\overline{w}_{1d}$. The modified hypothesis rows indicate the conditions under which deletion of $\overline{w}_{1d}$ is needed. The first row is chosen because it is assumed that each template is generated by the view designer in such a way that row matching can be pre-determined. Therefore, the run-time characteristics of $\overline{w}_{1d}$ can be specified at design time — see discussion of this problem at the end of the example in Section 3.2.

$De(\{t\}_i, R_i, force) : \Delta_i = [\{t\}_i \cup \{\overline{w}_{1d}\}]$. For either partial or full templates, $\overline{w}_{1d}$ is needed only in the case where no hypothesis row is eliminated as a direct result of applying substitution rules, and all hypothesis rows are represented in the instance.

Example: In the previous example, forced deletion of (ABCD) results in a tableau with rows (ABCD) and $\overline{w}_{1d} = (\mathrm{A}b_1\mathrm{C}d_1)$. The latter only needs to be deleted if no tuple of the form $(\mathrm{ABC}d_3)$ exists and tuples of the form $(\mathrm{AB}c_1d_2)$ exist. On the other hand, deletion of (AbCd) would require no further deletions, since it subsumes the (substituted) first row.

For *changes,* each template row w_j is duplicated, resulting in $w_j \leftarrow (w_j, [w_j])$. The conclusion row thus formed is replaced by $(\{t\}_i, [\{t'\}_i])$, which is pushed up the template, so that the variables in $\{t\}_i$ will replace those in w_j, and the variables in $\{t'\}_i$ will replace the corresponding variables in $[w_j]$. If no hypothesis row contains the same expression as the conclusion row, further changes may be needed. This is simulated by using the first row $(\overline{w}_k, [\overline{w}'_k])$ for which $\overline{w}_k \neq \overline{w}'_k$, denoted by $w_{1c} = (\overline{w}_{1c}, [\overline{w}'_{1c}])$. If no such row exists, then the change does not violate the constraint. This set of changes is followed by deriving insertions that may be needed as a consequence of the appearance of new tuples described by $\{t\}_i$ and $\overline{w}_{1c}$. $Ch(\{t,t'\}_i, R_i, force) : \Delta_i = (\{t\}_i, [\{t'\}_i]) \cup (\overline{w}_{1c}, [\overline{w}'_{1c}]) \cup In(\{t\}_i, R_i, force) \cup In(\overline{w}_{1c}, R_i, force)$.

Example: In the previous example, where $R_1 <$ABCD,{A→→B|C}, the request $(Ch, (\mathrm{ABcD}, [\mathrm{AB'cD}]), R_1, force)$ generates the substituted template

A	B	C	D	[A	B	C	D]
(A	b_1	c_2	d_1,	[A	b_1	c_2	d_1])
(A	B	c_1	d_2,	[A	B'	c_1	d_2])
(A	B	c_2	D,	[A	B'	c_2	D])

where the second row corresponds to $w_{1c} = (\overline{w}_{1c}, [\overline{w}'_{1c}])$. Insertion of (ABc_1d_2) and (ABc_2D) may require insertion of $(Ab_2c_2D_i)$ and (ABc_2D_j). In fact, these expressions describe tuples which already exist in the relation, due to the fact that the relation must have been consistent with respect to the template dependency before the update, and thus these insertions are not needed. Therefore, only the second row and the conclusion row in the template need to be stored in the underlying modification table.

d) R_i is subject to functional dependencies and a template dependency

For deletions, since they do not affect functional dependencies, the rule is performed as in (c), and thus $De(\{t\}_i, R_i, force) : \Delta_i = [\{t\}_i \cup \overline{w}_{1d}]$.

For insertions or changes, the updates are assumed to be processed in the order: preserve the functional dependencies first, and then preserve the template dependency. With this processing order, if tuples are to be deleted in order to preserve a functional dependency, they will not contribute to determining additional updates needed to satisfy the template dependency.

First, pairs $(w_{fd},[w'_{fd}])$ resulting from forcing functional dependencies are generated as described in (b) — for relations subject to functional dependencies only. The initial expression used to check a functional dependency corresponds to the conclusion row of the template dependency. The execution of the updates in $(w_{fd},[w'_{fd}])$ may then require an additional set of insertions and deletions in order to maintain the consistency of the template dependency. Insertion rules for template dependencies are applied for $\{t\}_i \cup \{w_{fd}\}$, and the resulting insertions $\{(\overline{w}_{s+1})_j\}$ may require further deletions in case of disagreement over functional dependencies. Finally, all deletions must be processed against the template dependency.

For insertions and changes, the interaction between a simple (Y-partial) template dependency and a set of functional dependencies that affect Y may cause conflicting updates, where insertions *disagree* over some functional dependency.

Example: Consider R_1<ABCD,{BC→AD,*[ABD, AC, CD]}>. The join dependency corresponds to the template

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_3	c_2	d_1
a_1	b_1	c_2	d_1

Forced insertion of $\{t\}_1$=(ABCD) generates the pairs (ABCD,[A'BCd_1]), and (ABCD,[a_1BCD']) to preserve the functional dependency, and the expressions $(\overline{w}_{s+1})_j = \{(ABc_2D), (Ab_1Cd_1), (a_1b_1CD)\}$ to preserve the template dependency. Let the domain of each attribute be a set of digits, and (ABCD) = (0000), and assume r_1 contained tuples {(0499), (5309), (0485), (6205)} before the insertion. This means that $(w_{fd},[w'_{fd}]) = \emptyset$, and that (Ab_1Cd_1)= {(0409) — because of the two first tuples; and (0405) — because of the two last tuples}. Thus both (0409) and (0405) must be inserted to satisfy the template dependency, and yet together they violate the functional dependency. Hence, further information would have to be provided to disambiguate the update requested. In fact, rather than allowing for more input, cases such as these are not accepted as valid update requests by the update validation algorithm.

As will be proved in Chapter 4, there are identifiable cases where this problem does not occur. Let X→A be a functional dependency whose attributes are represented in the conclusion row of an Y-partial simple template dependency by the symbols $\overline{x}\ \overline{a}$, where XA $\subseteq$ Y. The *allowed* interactions between this template dependency and functional dependency, for forced insertions and changes,

are in general those said to be in
class (HXA): at least one hypothesis row contains the symbols $\bar{x}\ \bar{a}$
If there is at most one non-trivial BCNF partition, then the set of allowed interactions is extended to include also dependencies in
class (A): all hypothesis rows contain the symbol $\bar{a}$,
class (HX): the symbols in $\bar{x}$ appear in at least one hypothesis row.

The rules $In(\{t\}_i, R_i, force)$ are hence processed in the order
1. (Preliminary functional dependency housekeeping) Process the insertion as for functional dependencies only — i.e., through changes $\{(w_{fd}, [w'_{fd}])\} = (\Delta I(t_{FD}), \Delta D(t_{FD}))$.
2. Push $\{t\}_i \cup \Delta I(t_{FD})$ down the template, generating $\{(\bar{w}_{s+1})_j\}$
3. If there is more than one non-trivial BCNF partition, the only interaction allowed is class (HXA), which requires no further deletions. In this case, all inserted tuples $(\bar{w}_{s+1})_j$ are "Good" (in the sense of not requiring further deletions). The set of good insertions is denoted $\{(\bar{w}_{s+1})_j^G\}$.
4. If there is only one partition, and it interacts with the template dependency under classes (A) or (HX) (which obviates the need to check for (HXA)), eliminate from r_i all tuples that cause functional dependency violation, given the insertions $\{(\bar{w}_{s+1})_j\}$.
5. Push the deletions in (1.) and (4.) up the template to ensure these deletions are effectively achieved.
The set of deletions performed in (4.) and (5.) is denoted $\{(\bar{w}_{s+1})_j^B\}$ (i.e., those that cause "BAD" interactions between tuples to exist, given $\{(\bar{w}_{s+1})_j\}$).

$In(\{t\}_i, R_i, force) \ : \ \Delta_i \ = \ \{t\}_i \cup \Delta I(t_{FD}) \cup \{(\bar{w}_{s+1})_j^G\} \ - \ [\Delta D(t_{FD}) \cup \{(\bar{w}_{s+1})_j^B\}]$.

Similarly, $Ch(\{t,t'\}_i, R_i, force) : \Delta_i = (\{t\}_i, [\{t'\}_i]) \cup (\bar{w}_{1c}, [\bar{w}'_{1c}]) \cup \Delta_i' \cup \Delta_i''$, where
$In(\{t\}_i, R_i, force) : \Delta_i'$ and
$(In(\bar{w}_{1c}, R_i, force)) : \Delta_i''$.

Example: Let R_1<CMS,{C→→M|S, MS→C}> (this is the same scheme R_1 of the example in Section 3.2), which exhibits the class(A) interaction (c_1 appears in every row)

C	M	S
c_1	m_1	s_1
c_1	m_2	s_2
c_1	m_2	s_1

Forced deletion of (CMS) generates the rows (CMS) and $\bar{w}_{1d}$ = (Cm_1S).

Forced insertion of (CMS) is processed as follows:
1. $(w_{fd}, [w'_{fd}])$ = (CMS,[C'MS]) (N.B. in this case $\Delta I(t_{FD}) = \{t\}_i$);
2. (CMS) is then processed as an insertion for template dependencies and the insertions $\{(\bar{w}_{s+1})_j^G\}$ generated are (Cm_2S) and (CMs_1);
3. These additional insertions must be examined to see if the updated relation violates the functional dependency. In other words, existing tuples of the form (C'' m_2 S) and (C''' M s_1) (where $C'', C''' \neq$ C) may have to be deleted.
4. Finally, ($C'MS$), (C'' m_2 S) and (C''' M s_1) alone may not be deletable because of the template dependency. Therefore, these deletions are processed

using rules for template dependencies:
$\forall\ m_1,s_2\ [C'm_1S]$ — deleted if (C'MS) is deleted and $(C'Ms_2) \in r_1$;
$\forall\ m_1,s_2\ [C''\ m_1\ S]$ — deleted if $(C''\ m_2\ S)$ is deleted (in 3.) and $(C''\ m_1\ s_2) \in r_1$;
$\forall\ m_1,s_2\ [C'''\ m_1\ s_1]$ — deleted if $(C'''\ M\ s_1)$ is deleted (in 3.) and $(C'''\ M\ s_2) \in r_1$;
The final set of entries in the underlying modification table is

C	M	S	[C	M	S]
(C	M	S	[C'	M	S])
			[C'	m_1	S]
C	m_2	S	[C''	m_2	S]
			[C''	m_1	S]
C	M	s_1	[C'''	M	s_1]
			[C'''	m_1	s_1]

and the corresponding database state table records the relation state description for each update.

3.4.2. Valuation of system-generated variables

Run time valuation of placeholder variables presents no problems. Valuation of system-generated parametric variables needs more attention, to ensure no constraints are violated.

Parametric variables that result from select operators can be assigned any value from the specified selection range. Similarly, in the case of additional insertions where there are partial template dependencies, system-generated parametric variables can be assigned any value from a pre-defined set, for example from a set of marked nulls or system defaults. The coexistence of functional dependencies and partial template dependencies creates the problem of determining the value of system-generated parametric variables for additional insertions $\{(\overline{w}_{s+1})_j\}$ when these variables are subject to functional dependencies. The procedure that follows shows one way in which the system can assign values to such variables at run time so that no functional dependency is violated.

Consider a scheme R_i in Partitionable BCNF. Let TRIV denote a trivial partition, and consider a Y-partial template dependency (i.e., the attributes in R_i-Y are *not* subject to the template dependency). The template dependency interacts with the functional dependencies if there exists some functional dependency X→A such that $Y \cap (XA) \neq \emptyset$. Otherwise, no interaction occurs.
a) If there exists a determinant K such that K→X, and whose value is determined by the template dependency, then K→A and therefore the values of (XA) are determined by the rules below but considering the dependency K→XA. Otherwise,
b) If $(XA) \cap Y = \emptyset$, choose some existing value for A and then choose some existing X such that X→A.
c) If $R_i\text{-Y} \subseteq$ TRIV then values are assigned as if no functional dependency existed since all values subject to functional dependencies are determined by the template dependency (insertion rules for template dependencies);
Consider a functional dependency X→A that interacts with Y.
d) Case of
: $X \cap (R_i\text{-Y}) \neq \emptyset$ and $A \in R_i$-Y, choose A and then choose X;

: X ∩ (R_i-Y) ≠ ∅ and A ∈ Y, choose some existing X such that X→A;
: XA ∩ Y = X, and the value assigned to X by the template dependency already exists, so choose A such that X→A; For the last two cases above, if the value of X does not exist, choose a new value that does not violate the dependency.

3.4.3. Generation of the initial tableau T_1

Having determined the change Δ_i to each underlying relation r_i, it is now possible to determine its effects on the view. Each underlying update $u_i = (Op_i, \{t\}_i, R_i, exception)$ is processed separately, to allow derivation of its isolated contribution to the view. This is achieved by processing, for each underlying update, a sequence of tableaux $T_1 \cdots T_k$. The first tableau T_1 describes the initial modifications caused by this update; the rows of the final tableau T_k indicate the isolated update's possible effects on the view.

For each underlying operation $u_i = (Op_i, \{t\}_i, R_i, exception)$, the rows of the tableau T_1 are:
- the set Δ_i, taken from the underlying modification table, and
- for each remaining scheme $R_{j \neq i}$, the conclusion row w_{s+1} of the template for R_j or a placeholder expression if there is no template for R_j.

The expressions for each relation scheme should be independent of those for other schemes, with a different variable for each attribute, so that, for each attribute, there should be as many different (sets of) variables as there are relation schemes that contain that attribute.

Example: Let R_1<AB,{A→B}>, R_2<BC,{B→C}> undergo the set of updates (*In*,AB,R_1,force) and (*De*,BC,R_2,cond). The underlying modification table contains the entries

R_1: (AB,[AB'])
R_2: [BC].

Thus the tableau T_1 for isolated processing of the insertion in r_1 is

A	**B**	**[A**	**B]**	**C**
A	B	[A	B']	
	b_2			c_2

and the tableau T_1 for the deletion in r_2 is

A	**B**	**[B**	**C]**
a_1	b_1		
		[B	C]

3.4.4. Initialization of the database state table

For each tableau T_1 built from the underlying modification table, a new database state table is generated, where each entry is associated with a tableau row. The rules for generating these entries, $State(Op_i)$, indicate the conditions for which each row in Δ_i is inserted, deleted or changed, as well as which tuples in $R_{j \neq i}$ can be combined with the expressions in Δ_i to result in view deletions, insertions or changes. It is assumed that no duplicate tuples will be inserted into any relation.

Let $T_1 \cdots T_k$ be the tableau sequence built to process the isolated outcome of $u_i = (Op_i, \{t\}_i, R_i, exception)$. For each $R_{j \neq i}$, if the entry for R_j in the underlying modification table involves tuples of the form $[\{t\}_j]$ (i.e., $\{t\}_j$ describes a set of tuples that will be deleted from R_j), then the database state table entry associated with the corresponding row in T_1 indicates that $\{t\}_j$ *cannot* contribute to any view tuple insertion. In other words, no view tuple can be created using $\{t\}_j$, since these tuples will be deleted during the execution of the complete policy. In the previous example, for instance, the set of tuples (b_2c_2) = (BC) cannot participate in any view tuple insertion.

The rules $State(Op_i)$ are defined as depending on the existence (or absence) of a set of tuples in the relation. The database state table is initialized with the relation state description. Its derivation is accomplished as follows.

Conditions needed for maintaining an X-partial template dependency

For forced insertions:

Repeat for each hypothesis row w_j

a) replace w_j by the expression describing tuples to be inserted (e.g., $\{t\}_i$)

b) apply substitution inference rules to the template

c) the resulting conclusion row $(\overline{w}_{s+1})_j$ must be inserted if all hypotheses $\{\overline{w}_1 \cdots \overline{w}_{j-1}\ \overline{w}_{j+1} \cdots \overline{w}_s\}$ hold at the same time (i.e., $\bigwedge_j (\overline{w}_j) \in r$).

For conditional insertions:

Repeat for each hypothesis row w_j

a) replace w_j by the expression describing tuples to be inserted (e.g., $\{t\}_i$)

b) apply substitution inference rules to the template

c) no additional row $(\overline{w}_{s+1})_j$ can be inserted (i.e., $\bigwedge_j (\overline{w}_j) \in r \Rightarrow (\overline{w}_{s+1})_j \in$ r)

Thus the condition for inserting $\{t\}_i$ is a conjunctive expression where each clause is formed by the expressions obtained in (c) (i.e., no additional row is inserted).

For forced deletions

a) replace w_{s+1} by $\{t\}_i$ and push $\{t\}_i$ up the template

b) the additional row $\overline{w}_{1d}$ must be deleted if

$(\nexists\ \overline{w} \mid \overline{w} \in$ r-$\{t\}_i \wedge \Pi_X \overline{w} = \Pi_X \{t\}_i) \wedge (\bigwedge_j (\overline{w}_j \in r) \wedge \nexists j \mid \overline{w}_j \subseteq \{t\}_i)$.

For conditional deletions

a) replace w_{s+1} by $\{t\}_i$ and push $\{t\}_i$ up the template

b) $\{t\}_i$ is deleted if

$(\exists\ \overline{w} \mid \overline{w} \in$ r-$\{t\}_i \wedge \Pi_X \overline{w} = \Pi_X \{t\}_i\ ) \vee (\exists$ j $\mid \overline{w}_j \subseteq \{t\}_i\ ) \vee (\ \bigvee_j (\overline{w}_j \notin$ r)
)

For changes, impose the existence of the tuples to be changed, whose marked attributes will be modified, and apply rules for deletion of $\{t'\}_i$ (and $\overline{w}'_{1c}$), followed by insertion of $\{t\}_i$ (and $\overline{w}_{1c}$).

Conditions needed for maintaining functional dependencies

In case of conditional insertions (or changes), the functional dependency X→A must not be violated (i.e., no tuple of the form $(XA'z)$ may exist).

In case of forced insertions or changes given X→A, where some of the attributes in XA have been assigned parametric variables, all tuples of the form $\exists\ z \mid (XA'z) \in$ r, must be replaced by tuples of the form (XAz). For forced insertions (or changes) when there exist functional dependencies and a template dependency, performing additional insertions $(\overline{w}_{s+1})_j$ may require deletion of tuples from r_i if some valid valuation of $(\overline{w}_{s+1})_j$ and some tuple in r_i violate X→A. For forced changes, $\overline{w}_{1c}$ must agree with $\{t\}_i$ over XA.

Example: Let R_1<ABCDE,{A→→B|C, A→D}>. The dependencies can be represented by

A	B	C	D	E
a_1	b_1	c_2	d_1	e_1
a_1	b_2	c_1	d_1	e_2
a_1	b_2	c_2	d_1	e_3

As explained in Section 3.4.1, forced insertion of (ABCDE) requires insertion of (Ab_2CDE_1) and (ABc_2DE_2) (where E_1 and E_2 are values assigned by the system). These expressions are determined by replacing w_1 (and w_2) by (ABCDE), as follows

A	B	C	D	E
A	B	C	D	E
A	b_2	c_1	D	e_2
A	b_2	C	D	E_1

A	B	C	D	E
A	b_1	c_2	D	e_1
A	B	C	D	E
A	B	c_2	D	E_2

The associated relation database states are
(Ab_2CDE_1) (is inserted if): $\exists\ b_2,c_1,e_2 \mid (\mathrm{A}b_2c_1\mathrm{D}e_2) \in r_1$
(ABc_2DE_2) (is inserted if): $\exists\ b_1,c_2,e_1 \mid (\mathrm{A}b_1c_2\mathrm{D}e_1) \in r_1$.

Furthermore, to ensure the functional dependency is not violated, these two rows must be preceded by the entry (Ab_2c_2De_3, [Ab_2c_2D'e_3]), with the database state description
$\exists\ b_2,c_2,e_3,D' \mid D \neq D' \wedge (\mathrm{A}b_2c_2D'e_3) \in r_1$.

Conditional insertion of ABCDE uses the same set of template substitutions and indicates
(ABCDE) (is inserted if):
$\forall\ b_2,c_2,e_3,D'\ (\ D \neq D' \Rightarrow (\mathrm{A}b_2c_2D'e_3) \notin r_1)$ and
$[(\exists\ b_2,c_1,e_2,e_3) \mid (\mathrm{A}b_1c_2De_2) \in r_1)] \Rightarrow (\mathrm{A}b_2\mathrm{CD}e_3) \in r_1$ and
$[(\exists\ b_1,c_2,e_1,e_3) \mid (\mathrm{A}b_1c_2De_1) \in r_1)] \Rightarrow (\mathrm{AB}c_2\mathrm{D}e_3) \in r_1$.

Deletion of ABCDE uses the substituted template

A	B	C	D	E
A	b_1	C	D	e_1
A	B	c_1	D	e_2
A	B	C	D	E

Forced deletion of (ABCDE) may require deletion of $\overline{w}_{1d}$, i.e.:
(ABCDE) (is deleted) and
$\forall\ b_1,e_1$ (Ab_1CDe_1) (is deleted if)
$(\nexists\ e_3 \mid (\mathrm{ABCD}e_3) \in r_1) \wedge (\exists\ b_1,c_1,e_1,e_2 \mid ((\mathrm{AB}c_1\mathrm{D}e_2) \in r_1 \wedge (\mathrm{A}b_1\mathrm{CD}e_1) \in r_1))$.

Conditional deletion of (ABCDE) results in
(ABCDE) (is deleted if)
$(\exists\ e_3 \mid (\mathrm{ABCD}e_3) \in r_1) \vee (\forall\ e_1, b_1, (\mathrm{A}b_1\mathrm{CD}e_1) \notin r_1) \vee (\forall\ e_2, c_1, (\mathrm{AB}c_1\mathrm{D}e_2) \notin r_1)$.

When functional dependencies and a template dependency interact and there is more than one non-trivial partition, valid valuations of the database state description *cannot* include tuples that have been eliminated in the preliminary functional dependency housekeeping, and *should* include those which are inserted at that stage. In other words, the database state description should refer to the state $r_i \cup \{t\}_i \cup \Delta I(t_{FD}) - \Delta D(t_{FD})$ rather than to $r_i \cup \{t\}_i$.

Example: Let R_1<ABCD, {A→B, C→D, *[AB,BC,CD]}>, with the template

$$\begin{array}{cccc}
\mathrm{A} & \mathrm{B} & \mathrm{C} & \mathrm{D} \\
\hline\hline
a_1 & b_1 & c_1 & d_1 \\
a_2 & b_1 & c_2 & d_2 \\
a_3 & b_2 & c_2 & d_3 \\
\hline
a_1 & b_1 & c_2 & d_3
\end{array}$$

Forced insertion of (ABCD) generates the following changes due to functional dependency housekeeping:
$(\mathrm{ABCD},[\mathrm{AB'CD'}]) : \exists\ \mathrm{B'} \neq \mathrm{B}, \mathrm{D'} \neq \mathrm{D} \mid (\mathrm{AB'CD'}) \in r_1$
$(\mathrm{AB}c_2d_3,[\mathrm{AB'}c_2d_3]) : \exists\ c_2, d_3, \mathrm{B'} \neq \mathrm{B} \mid (\mathrm{AB'}c_2d_3) \in r_1$
$(a_1b_1\mathrm{CD},[a_1b_1\mathrm{CD'}]) : \exists\ a_1, b_1\ , \mathrm{D'} \neq \mathrm{D} \mid (a_1b_1\mathrm{CD'}) \in r_1$.

For instance, one of the insertions generated by pushing $(\mathrm{AB}c_2d_3)$ down the template is
$(\mathrm{AB}\bar{c}\bar{d}) : \exists\ a_2, \bar{c}, d_2, a_3, b_2, \bar{d} \mid (a_2\mathrm{B}\bar{c}d_2) \in \bar{r}_1 \wedge (a_3b_2\bar{c}\bar{d}) \in \bar{r}_1$,
where $\bar{r}_1 = r_1 \cup$ (ABCD) $\cup$ $(a_1b_1\mathrm{CD})$ $\cup$ $(\mathrm{AB}c_2d_3)$ - (AB'CD') - $(\mathrm{AB'}c_2d_3)$ - $(a_1b_1\mathrm{CD'})$.

Restricting the set of valid valuations to this intermediate relation state $\bar{r}_1$ guarantees that no additional insertion violates any functional dependency. Due to the characteristics of the class (HXA) interaction, the same restrictions on $\bar{r}_1$ can be equivalently described as
$(a_2B\bar{c}d_2) \in r_1 \wedge (a_3b_2\bar{c}\bar{d}) \in r_1$, where $(\bar{c} = C \Rightarrow (\bar{d}, d_2 = D)\ ) \wedge (a_3 = A \Rightarrow b_2 = B)$.

This takes the preliminary functional dependency housekeeping into account, and the database state description does not need to consider intermediate relation states. This allows the output to remain in terms of valuations over $r_i \cup \{t\}_i$.

3.4.5. Rules for execution of the view generating function

The $Ex(q_v)$ rules provide the transition from tableau T_j to T_{j+1}, in the sequence $T_1 \cdots T_k$. This transition is achieved in two steps. First, one operation of the view generating function is simulated, mimicking actual project, select and join operations over real tuples, in a manner similar to the rules applied when processing tableau queries. This creates the transition from T_j to T'_{j+1}. This is followed by what is called *functional dependency checking* over the rows of the new tableau T'_{j+1}, in order to determine possible changes from placeholder to parametric variables. This checking yields the tableau T_{j+1}. The rules for simulating an operation in q_v are as follows.

1) Projection $(\Pi_X\ R)$

Replace the rows representing R by rows where only the set of attributes X appear, omitting the remaining attributes.

2) Selection $(\sigma_{X_c}\ R)$ (where X_c is the range of values for the set of attributes X)

Replace the rows representing R by rows where each attribute appearing in X is replaced by a system-generated parametric variable. Store this variable and the corresponding range C in the database state table. If some attribute in X is already represented by a parametric variable A_i, its entry in the database state table will be $A_i \wedge C$.

3) Join $(R_a \times R_b)$

The rows to be joined can be of the types (w), [w'] or (w,[w']).

a) Joining rows containing a single symbolic expression

Each pair of rows $w_a \in R_a$ and $w_b \in R_b$ is replaced by a single row w_{ab}, where the common attributes $X = R_a \cap R_b$ are pairwise unified, and the remaining variables $Y_a=(R_a - X)$ and $Y_b=(R_b - X)$ are copied from w_a and w_b.

b) Joining with a row containing a pair of symbolic expressions

These rows originate from either forced insertions or changes, and are of the form w_i=(inserted tuples, [deleted tuples]). If w_a contains a single expression and w_b consists of a pair of expressions, w_{ab} consists of a pair of expressions, resulting from joining each component of w_b separately to the expression in w_a.

Execution of the join operator involves two parts: determining the final value for the unification, and modifying the database state description in the database state table corresponding to the new tableau row. The unification chooses a parametric variable over a placeholder variable, and a user-entered parametric variable over a system-generated parametric variable; when joining two placeholder variables (or two parametric system-generated variables), the one with the lower index is chosen.

Let A_a and A_b be the attributes to be joined into A_{ab}. The possible combinations of variables and the resulting join are as follows (recalling that capital characters stand for parametric variables, and lowercase characters denote placeholder variables):

A_a	A_b	A_{ab}	State description
a_1	a_2	a_1	$a_2 = a_1$
a_1	A	A	a_1=A
A	A_1	A	A_1= A (and all conditions for A_1)
A_1	A_2	A_1	$A_2 = A_1$ (and all conditions for A_2 and A_1)

Once the value for each unified variable in $w_a \times w_b$ is defined, the entries for w_a and w_b in the database state table are copied into a new entry describing w_{ab}, together with all conditions necessary for the join. This new entry is formed by the conjunction of the entries from w_a and w_b, together with any value assignments resulting from the join. If this conjunction results in an invalid logic clause, any output expression that is associated with this clause corresponds to an update which will not occur. If the operation being processed by the tableau involves an insertion — i.e., the initial tableau T_1 is based on $In(\{t\}_i, R_i, exception)$ or $Ch(\{t,t'\}_i, R_i, exception)$ — then the rows being joined must be checked to eliminate the tuples that cannot contribute to a view tuple insertion.

Finally, functional dependencies must be checked whenever they apply to attributes that are assigned parametric variables by a join. Consider $R_i<\{A_{i_j}\},\{C_{i_n}\}>$, where (X→A) $\in$ $\{C_{i_n}\}$. If joining two rows assigns a parametric variable to attribute X, then attribute A is also assigned a (system-generated) parametric variable.

Example: Let R_1<AB,{A→B}>, R_2<BC,{B→C}> undergo the set of updates (*In*,AB,R_1,force) and (*De*,BC,R_2,cond). The underlying modification table contains the entries

R_1: (AB,[AB′])
R_2: [BC].

Let $q_v=R_1\times R_2$. Processing (*In*,AB,R_1,force) results in the tableau T_1

A	**B**	**[A**	**B]**	**C**
A	B	[A	B′]	
	b_2			c_2

where the conditions associated with the second row are that tuple (b_2c_2) = (BC) *cannot* participate in any view tuple insertion. The condition for joining (AB) to (b_2c_1) is b_2=B. The output predicts deletion of $(AB'C'_1)$ (where $B'\to C'_1$), and insertion of (ABC_1) if $(BC_1 \in r_2 \wedge \mathrm{B}\to C_1 \wedge BC_1 \neq BC)$. This means, in fact, that the insertion will not occur, because (BC) will be deleted by the operation in R_2. Thus, the only possible result of this insertion is to delete tuples of the form $(AB'C'_1)$ from the view! The deletion of (BC) causes deletion of all tuples (a_1BC) from the view.

3.5. Complete example revisited

This section shows the same example as Section 3.2 in symbolic terms. Let R= {R_1<(C,M,S),{C→→M|S, (MS)→C}>, R_2<(C,P), {C→P}>}, and a view defined by q_v = $R_1\times R_2$. The request to be processed is
<*In*, CMSP, {(*In*,CMS,R_1,*force*), (*In*,CP,R_2,*cond*)}>.

The forced insertion in R_1 is the example shown for forced insertion of a tuple in a relation subject to a template dependency and a functional dependency interacting according to class (A), in Section 3.4.3d, and will not be repeated here. Conditional insertion of (CP) into r_2 can only be accomplished if this relation does not contain a tuple (CP′). The final set of entries in the underlying modification table is

C	**M**	**S**	**[C**	**M**	**S]**	**P**
(C	M	S	[C'	M	S])	
			[C'	m_1	S]	
C	m_2	S	[C''	m_2	S]	
			[C''	m_1	S]	
C	M	s_1	[C'''	M	s_1]	
			[C'''	m_1	s_1]	
C						P

Isolated contribution of r_2, for instance, uses the tableau T_1

C	M	S	P
c_1	m_2	s_1	
C			P

where $(c_1m_2s_1)$ is the conclusion row for the template dependency C→→M|S. The conditions associated with this expression are that it cannot describe the tuples that are deleted from r_1, e.g., $(c_1m_2s_1) \neq (C'\text{MS})$. The execution of the view generating function joins these two rows into (Cm_2s_1P), which is the only row of tableau $T_k = T_2$, the last tableau of the sequence for updating r_2. This join can only be accomplished if c_1=C. Thus, $(\text{C}m_2s_1\text{P})$ appears in the view if the associated entry in the database state table holds, i.e.,

$$\{t\} = (c_1m_2s_1) \in r_1 \wedge (c_1{=}\text{C})$$

$\wedge$ {t} does not describe the tuples deleted from r_1.
The isolated contribution of Δ_1 to the view is generated in a similar way, starting from a tableau T_1 that contains all but the last row of the underlying modification table. This last row is replaced by (c_2s_3), representing the tuples of r_2.

The final set of output rows is

Isolated contribution of r_1
$(\text{CMS}P_1, [C'\text{MS}P_1])$
$[C'm_1\text{S}P_2]$
$(\text{C}m_2\text{S}P_1, [C''\ m_2\text{S}P_3])$
$[C''\ m_1\text{S}P_3]$
$(\text{CM}s_1P_1, [C'''\ \text{M}s_1P_4])$
$[C'''\ m_1s_1P_4]$

Isolated contribution of r_2
$(\text{C}m_2s_1\text{P})$

where the database state descriptions associated with the deletion $[C'm_1\text{S}P_2]$, for instance, are
: $\forall\ m_1,s_2$ $(C'\text{MS}) \in r_1$ — i.e., (C'MS) must have been deleted
: $(C'\text{M}s_2) \in r_1$ (both conditions correspond to the relation state description for r_1, computed by $State(Op_i)$);
The execution of the join in q_v adds the state description
: $(C'P_2) \in r_2$
Finally, functional dependency checking determines the last constraint on the database state description
: $C' \rightarrow P_2$.

A valid valuation for $(C'm_1SP_2)$ is (Phys, Wed 10, James, Prof1). For the instance given in Section 3.2, there is no valid valuation for the deletions $[C''\ m_2SP_3]$ and $[C''\ m_1SP_3]$. These deletions correspond to a situation requiring (Course≠Chem, Fri 15, James) $\in r_1$.

3.6. The update validation algorithm and update chains

The update validation algorithm borrows some concepts from traditional tableau manipulation. Non-distinguished and distinguished variables approximately correspond to placeholder and parametric variables, respectively; as in tableau queries, blanks and constants are allowed. The final set of rows represents changes to the view, and is obtained in a manner similar to that of derivation of a summary row in a (tagged) tableau query. Unlike summary rows, they do not represent all tuples in the view, and may contain placeholder (non-distinguished) variables.

Checking of functional dependencies after each join is similar to applying chase with F-rules, as it transforms placeholder (non-distinguished) variables into parametric (distinguished) variables. Unlike traditional tableaux, a column can contain more that one parametric (distinguished) variable, as long as each variable is used in a set of rows that corresponds to different underlying conditions — see example in Section 3.5.

The generation of Δ_i expressions has an analogy with tableau chase computation as well. Let R_i be subject to a template dependency. Consider any mapping σ which maps template rows to an underlying relation. If this mapping is a symbolic valuation generated by replacing some template hypothesis w_k by a symbolic tuple expression, the conclusion row obtained describes an insertion. If this mapping is generated by replacing the conclusion by a symbolic expression, the substitution corresponds to enforcing a deletion. If the template dependency is not simple, the insertion expression $(\overline{w}_{s+1})_k$ or the deletion expression $\overline{w}_{1d}$ thus obtained must be checked against the template to ensure that there is no valuation for which their insertion (deletion) requires in turn insertion (deletion) of additional tuples. It is shown in Chapter 4 that this cannot happen with simple dependencies. This substitution mechanism is iterated until all possible insertion (deletion) expressions are computed. Such a ripple effect will be called the generation of an *insertion (or deletion) chain.* It will be assumed that each mapping generated in a chain is created over a new copy of the template dependency, whose symbols have not appeared before in the chain.

The creation of a chain can be compared to chasing a tableau using template dependency T-rules: in the update validation algorithm, the tableau being chased is in fact a relation composed of symbolic tuple expressions, which is initially consistent with respect to the template dependency. An insertion introduces a new symbolic expression, and the tableau (relation) is chased to determine which new rows must be added to keep it consistent. Deletion (and change) processing is similar to reducing a tableau by row subsumption (see Chapter 2). Once the conclusion row is replaced and substitutions are applied, $\overline{w}_{1d}$ (w_{1c}) is the first row found that subsumes the conclusion.

The difference between the usual tableau chase and this type of iterative process is that traditional chases start from a fixed tableau (or relation), whereas here the rules $Op(\{t\}_i, R_i, force)$ must consider the set of all possible valid tableau (relation) configurations. Therefore, even though the chase process for a single template dependency is always finite, the process of generating insertion or deletion chains for non-simple templates may never end.

The term *substitution pattern* is used here to refer to the fact that any sequence of mappings in a chain is characterized by a pattern of "old" and "new" values. Old values refer to the symbols that are inherited by the next mapping in the chain; new symbols are introduced by using a new template copy at each new mapping. The pattern depends on how rows match each other.

Example: Let old values be represented by ○, and new values by ●. The template on the left is a template dependency; the one in the middle is the mapping generated by $w_3 \leftarrow ABC$; the one on the right describes the pattern that characterizes this substitution. Every time a mapping in a chain is generated by replacing row w_3, the set of old and new values can be visualized as forming this pattern.

A	B	C	A	B	C	A	B	C
a	b	c_1	a	B	c_1	●	○	●
a	b_1	c	a	b_1	C	●	●	○
a_1	b	c	A	B	C	○	○	○
a_1	b_1	c_1	A	b_1	c_1	○	●	●

For this template dependency, subsequent substitutions in other rows will *always* produce new insertion expressions in the conclusion row. *New insertion expressions* are those that cannot be derived from expressions that describe tuples that already exist in the relation (i.e., cannot be derived from database state descriptions).

For instance, if the conclusion (Ab_1c_1) were to replace the second row, the result would be a new insertion expression $(a'b_1c')$, which cannot be derived from the existing tuples described by the expressions in the previous template (rows 1 and 2) and in this new template (rows 1 and 3).

A	B	C
A	b'	c'
A	b_1	c_1
a'	b'	c_1
a'	b_1	c'

This second insertion is needed to maintain consistency only after insertion of (Ab_1c_1) is performed. If $(a'b_1c')$ were now to replace the third row, yet another new insertion would appear, $(a'b''c'')$, following the pattern shown for replacement of the third row.

A	B	C
a"	b_1	c"
a"	b"	c'
a'	b_1	c'
a'	b"	c"

Consider, for instance, a relation subject to this constraint, with the consistent extension {(001), (010), (122), (221), (313), (332)}. Forced insertion of (100) can only be accomplished if (111) is inserted as well (because of (001) and (010)). However, once (111) is inserted, (212) must be inserted also, because of tuples (122) and (221). If (212) is inserted, (233) must be inserted as well, because of (313) and (322). This is an instance where insertion of (100)=(ABC) would fall into the mapping sequence just indicated (generated by substitutions of w_3, w_2 and w_3). There is no limit to this process: as long as new mappings are generated, new expressions will appear. Any valuation that takes each new symbol into an attribute value that has not been chosen before will determine a different database instance, for which the new insertion will be needed to maintain consistency. This new insertion depends on the insertion that preceded it in the chain, and cannot be determined independently. If, for instance, the extension had been {(001), (010), (313), (332)}, insertion of (100) would only require insertion of (111) and no further updates would be needed. In other words, for non-simple template dependencies, insertion chains can be finitely described only if the extensional database is known.

The next chapter will prove the correctness of the update validation algorithm and determine its complexity. Some of the similarities between the update validation algorithm and tableau chases will help the development of the proofs. Insertion and deletion chains will be accomodated, as it will be proved they have length one for simple templates. More general chains will be investigated in Chapter 6.

3.7. Summary outline of the validation algorithm

Execute steps 1 and 2, stopping with the appropriate message if an error is found in the input. Record all changes in variables in the database state table entries.

STEP 1

CHECK INPUT CONSISTENCY AND PREPROCESS CONSTRAINTS

Let the input be $<R, q_v, u_v>$, where
$u_v = <OP, \Gamma_v, \{u_i = (Op_i, \{t\}_i, R_i, exception)\}>$

1. For each relation scheme R_i DO

1.1. If there are functional dependencies, build the BCNF partitions. If this is not possible, interrupt execution with an error message.

Let Npart be the number of non-trivial BCNF partitions, and TRIV denote a trivial partition (i.e., where no functional dependency exist).

1.2. If $\{C_{i_n}\}$ contains a non-simple template dependency, stop with error message (not a simple template dependency)

1.3. If {Npart=0} OR
$\{\{C_{i_n}\}$ does not contain a template dependency$\}$ OR
$\{Op_i = De\}$ OR
$\{exception = cond\}$ OR
$\{Op_i = Ch$ and all attributes being changed belong to TRIV $\}$ OR
{there is a Y-partial template dependency and Y $\subseteq$ TRIV — the functional dependencies do not interact with the template dependency} then process next scheme R_i+_1 (go to OD)

N.B. 1.4 and 1.5 refer to the situation where $\{C_{i_n}\}$ contains both functional dependencies and a template dependency

1.4. If Npart=1, the functional dependencies in the partition can only interact with the template dependency according to classes (A) or (HX). If neither of these cases occur, stop with error message (ambiguous update); else, go to OD.

1.5. For j = 1 to Npart do

Consider each dependency X→A in partition(j). If the interaction between this

```
dependency and the template dependency is not (HXA), stop with error message.
OD (1.) STEP 2 (update validation algorithm)
DETERMINE ISOLATED CONTRIBUTIONS OF UPDATING EACH $r_i$
1. Build the underlying modification table
 For each request $u_i = (Op_i, \{t\}_i, R_i, exception) \in u_v$ do
    1.1. Compute $\Delta_i$ and store its expressions in the underlying modification
    table, together with the relation state description determined by $State(Op_i)$.
2. For each request $u_i = (Op_i, \{t\}_i, R_i, exception) \in u_v$ do
    2.1. Build the initial tableau $T_1$ based on the entries of the underlying modification
    table for $R_i$, and initialize a new database state table for the sequence.
    2.2. Initialize a set F, of functional dependencies to be checked at each step
      F = set of functional dependencies that apply to $R_i$
    2.3. Create the sequence $T_2 \cdots T_k$
      2.3.1. While there are operations to execute in $q_v$ do
      - perform the next operation in $q_v$, using $Ex(q_v)$ rules;
      - if the last operation was a join with scheme $R_j$ then do
        : F=F $\cup$ {functional dependencies in $R_j$}
        : determine additional parametric variables: if $X \to A \in F$, and X has
        been assigned specific parametric values M during the join, transform
        attribute A into parametric variable $A_j$, and add $M \to A_j$ to the data-
        base state table entry;
        : if $X \to A$ assigned a parametric variable to A, $F = F - \{X \to A\}$
      - if the last operation was a projection that eliminated the attribute A
      corresponding to the right side of some dependency $X \to A \in F$ then $F = F -$
      $\{X \to A\}$.
    2.4. Associated with each row of $T_k$ indicate the corresponding class of initial
    database states as appears in the database state table.
```

Chapter 4

Correctness and complexity of the update validation algorithm

This chapter is composed of two parts. In the first, the update validation algorithm proposed in Chapter 3 is proved correct; in the second, its complexity is analyzed.

4.1. Basic concepts and definitions

Let a template dependency be defined as $w_1 \cdots w_s | w_{s+1}$, and row w_j match row w_k over a set of attributes $\{a_M\}$. This is denoted by Same(j,k)=$\{a_M\}$, which is the notation used by Sadri and Ullman [SAD80].

Let $(\overline{w}_1 \cdots \overline{w}_s)|\overline{w}_{s+1}$ be a symbolic valuation of an X-partial template dependency, where $\overline{w}_{s+1}$ describes a set of possible insertions which may be needed to maintain the constraint described by the template. The expression $\overline{w}_{s+1}$ represents a ***necessarily redundant insertion*** if either

a) $\exists$ j | $(\overline{w}_{s+1})$ is equivalent to $\overline{w}_j$ (i.e., the tuples already exist in the relation), or

b) $\exists$ j | $\Pi_X(\overline{w}_{s+1})$ is equivalent to $\Pi_X \overline{w}_j$, (i.e., the tuples needed to preserve the embedded constraint already exist in the relation).

4.2. Proving the correctness of the validation algorithm

4.2.1. Outline of the proof steps

Theorem I: The update validation algorithm correctly predicts the set of all possible outcomes for arbitrary update policies u_v on SJP views, together with the class of initial database states for which each such modification becomes visible in the view.

Proof

Theorem I is proved by using the following Lemmas and Theorems, whose proofs appear in the next section.

Theorem 1: The set of expressions generated according to $Op(\{t\}_i, R_i, cond)$ describes a minimal set of updates which accomplishes the corresponding underlying operation Op and maintain the semantic integrity of r_i as defined by the constraints $\{C_{i_n}\}$.

Theorem 2: The set of expressions generated according to $De(\{t\}_i, R_i, force)$ describes a minimal set of deletions necessary to accomplish the corresponding underlying operation and maintains the semantic integrity of r_i as defined by the constraints $\{C_{i_n}\}$.

Theorems 3-4: (Similar to Theorem 2, for insertions and for changes.)

Theorem 5: The set of rules $State(Op_i)$, that determine the underlying conditions for a row to appear in each initial tableau T_1, exactly describe the relation state for which the corresponding set of tuples exist.

Theorem 6: The update validation algorithm always stops.

Theorem 7: The contribution to the SJP view V=(R,q_v) of the set of underlying tuples described in a tableau T_1 is exactly described by the final tableau T_k of the corresponding tableau sequence generated by the update validation algorithm.

Theorem 8: The database state table entries associated with each output row in a final tableau T_k in the update validation algorithm exactly describe the class of database states necessary for the corresponding update to be reflected in the view.

Theorem 9: For monotonic view functions, deleting a tuple t from an underlying relation extension r_i does not add tuples to the view extension; adding t to r_i does not delete tuples from the view extension.

Theorem 10: Any possible outcome for a complete update policy $u_v = \{(Op_i, \{t\}_i, R_i, exception)\}_{i=1..n}$ on an SJP view is derivable from the set described by the rows of $\{T_k\}_{i=1..n}$ representing the isolated contributions of the underlying updates $u_i = (Op_i, \{t\}_i, R_i, exception)$ to the same view.

Using these theorems, the proof goes as follows.

a) Correctness in interpreting an isolated update

o Theorems 1-4 (minimality and correctness of the set generated by $Op(\{t\}_i,R_i,exception)$) show that the set of underlying updates specified for relation r_i is correctly interpreted by entries in the underlying modification table, and that these entries describe a minimal set of updates that achieve the update requested and maintain the database consistency, for the allowed types of constraints in $\{C_{i_n}\}$.

o Theorem 5 (correctness of $State(Op_i)$) shows that the database state table entries associated with each row of a tableau T_1 describe the class of relation states for which the corresponding tuples contribute to view formation.

o Theorems 6 (finiteness of the update validation algorithm), and 7 (minimality and correctness of the view updates represented in T_k, the last tableau of a sequence) consider a tableau sequence $T_1 \cdots T_k$. They show that, unless the algorithm stops as a consequence of an error, the final tableau T_k always exists. This tableau exactly describes all view tuples which may originate from the initial set of underlying updates portrayed in T_1, the first tableau of the corresponding sequence.

o Theorem 8 (correctness of the final database state table entries) shows that each row in a tableau T_k is correctly associated with the description of the class of underlying database states for which the corresponding view update occurs.

Therefore, Theorems 1 through 8 show that, if any isolated underlying relation is updated by $u_i = (Op_i,\{t\}_i,R_i,exception)$, the update validation algorithm exactly provides as output the description of the isolated contribution of u_i to any SJP view, and the class of database states for which these updates occur.

b) Correctness in considering all possible consequences of a set of underlying updates

o Theorem 9 (composition of underlying updates) shows that underlying deletions or underlying insertions in different relations do not interact destructively, and therefore the order in which the underlying relations are updated does not influence the final view extension.

o Finally, Theorem 10 (sufficiency in checking isolated updates) shows that any view update resulting from a set of "simultaneously executed" underlying updates $\{u_i = (Op_i,\{t\}_i,R_i,exception)\}$ can be derived by composing the isolated contributions of u_i, as long as the view belongs to the class of SJP views.

Since by Theorems 1-8 the update validation algorithm correctly describes all effects of isolated updates, and since all updates to underlying relations are processed, then the set of final tableaux $\{T_k\}$ contains the description of all possible view updates that can result from the policy $u_v = \{(Op_i,\{t\}_i,R_i,exception)\}$ for SJP views. Furthermore, the associated database state table entries indicate exactly when these updates can occur.■

4.2.2. Proofs of Lemmas and Theorems

Theorem 1: The set of expressions generated according to $Op(\{t\}_i, R_i, cond)$ describes a minimal set of updates which accomplishes the corresponding underlying operation Op and maintain the semantic integrity of r_i as defined by the constraints $\{C_{i_n}\}$.

Proof

The expressions Δ_i generated are $In(\{t\}_i, R_i, cond)$: $\{t\}_i$, $De(\{t\}_i, R_i, cond)$: $[\{t\}_i]$ and $Ch(\{t,t'\}_i, R_i, cond)$: $(\{t\}_i, [\{t'\}_i])$. In other words, for conditional updates, the update performed involves exclusively the tuples indicated, and occurs only if no constraints are violated. Therefore, the proof of minimality and correctness for conditional updates is trivial. ■

Theorem 2: The set of expressions generated according to $De(\{t\}_i, R_i, force)$ describes a minimal set of deletions necessary to accomplish the corresponding underlying operation and maintains the semantic integrity of r_i as defined by the constraints $\{C_{i_n}\}$.

This theorem is proved by Lemmas 2.1 (sufficiency of $De(\{t\}_i, R_i, force)$) and 2.2 (minimality of $De(\{t\}_i, R_i, force)$).

Lemma 2.1: (sufficiency in executing the update while maintaining consistency) The set of deletions generated by $De(\{t\}_i, R_i, force)$ accomplishes the corresponding underlying operation, and does not violate the constraints $\{C_{i_n}\}$ for relation r_i.

Proof

When there are no constraints or functional dependencies only, the proof is trivial: deletion of $\{t\}_i$ is sufficient to obey the underlying request, and the relation is kept consistent. The proof that follows refers to maintaining template dependencies.

Consider an X-partial template dependency, and let $\bar{r}_i = r_i - \{t\}_i$. Let $\{t_{rem}\}$ describe a set of tuples in $\bar{r}_i$ such that $\Pi_X\{t\}_i$ is equivalent to $\Pi_x\{t_{rem}\}$. By the deletion rules, deletion of $\{t\}_i$ ought to be sufficient to maintain consistency if $\{t_{rem}\} \neq \emptyset$. Assume $\{t_{rem}\} \neq \emptyset$ and yet deletion of $\{t\}_i$ is not sufficient. This means there exists a tuple $t \notin \bar{r}_i$ and a valuation ρ such that $\bigwedge_j (\rho(w_j) \in \bar{r}_i \Rightarrow t \in \bar{r}_i$ (i.e., $\bar{r}_i$ is not consistent). In other words, t is described by $\{t\}_i$, since the relation was consistent before the deletion. Since only the set X is subject to the template dependency,

$$\Pi_X(w_1) \wedge \cdots \wedge \Pi_X(w_s) \Rightarrow \Pi_X(w_{s+1}).$$

and $\Pi_X\{t\}_i$ describes $\Pi_X t$. However, this means $\Pi_X t$ is described by $\Pi_X\{t_{rem}\}$, and if $t \notin \bar{r}_i$ then $\{t_{rem}\} = \emptyset$, which contradicts the initial hypothesis. Thus, deletion of $\{t\}_i$ is sufficient if $\{t_{rem}\} \neq \emptyset$.

Suppose now $\{t_{rem}\} = \emptyset$. In this case, $De(\{t\}_i, R_i, force) : \Delta_i = \{t\}_i \cup \overline{w}_{1d}$. Proving sufficiency requires proving that the set $[\{t\}_i \cup \overline{w}_{1d}]$ achieves the deletion of $\{t\}_i$ and does not imply new deletions, i.e., these deletions can be achieved without causing deletion of any further tuples. Obviously, deletion of $\{t\}_i$ achieves the desired operation. The remainder of the proof refers therefore to the fact that deletion of $\{t\}_i \cup \overline{w}_{1d}$ is sufficient to maintain the consistency.

Let $\overline{w}_{1d}$ be chosen from some mapping σ, generated by $\sigma\colon w_{s+1} \leftarrow \{t\}_i$, where

$$\sigma(w_j)=\overline{w}_{1d} \wedge [\sigma(w_1)\wedge \ldots \wedge \sigma(w_s) \Rightarrow \{t\}_i].$$

Assume, by contradiction, that the set $\{t\}_i \cup \overline{w}_{1d}$ is not sufficient — i.e., other tuples must be deleted as well if consistency is to be maintained. If deletion of $\{t\}_i \cup \overline{w}_{1d}$ is not sufficient, then there exists some symbolic valuation ϕ that applies to the tuples described by $(r_i - [\{t\}_i \cup \overline{w}_{1d}])$ such that

$$\phi(w_1)\wedge \ldots \wedge \phi(w_s) \Rightarrow \overline{w}_{1d} \wedge \nexists\ k \mid \phi(w_k) \text{ is subsumed by } \overline{w}_{1d}.$$

Consider the mappings σ and ϕ, where

$$\sigma\colon w_{s+1} \leftarrow \{t\}_i \wedge \overline{w}_{1d}=\sigma(w_j)$$
$$\phi\colon w_{s+1} \leftarrow \overline{w}_{1d}.$$

Let R_i be defined over the set of attributes Y. For X-partial simple templates,

$$\forall\ w_i,w_j,\ \Pi_X\ \mathit{S}\text{ame(i,j)} \subseteq \Pi_X\ \mathit{S}\text{ame(j,s+1)} \wedge \Pi_{(Y-X)}\ \mathit{S}\text{ame(j,s+1)} = \varnothing.$$

Let J1=*S*ame(j,s+1) and NotJ1=Y-J1. Row $\phi(w_j)$ matches $\phi(w_{s+1})=\overline{w}_{1d}$ over exactly J1 and thus has different symbols for attributes in NotJ1; $\overline{w}_{1d}=\sigma(w_j)$ matches $\sigma(w_{s+1})=\{t\}_i$ over J1 and has different symbols for attributes in NotJ1. Therefore, the rows $\{\sigma(w_i)\}_{i\neq j}$ match $\overline{w}_{1d}$ over subsets of J1 and have distinct symbols for the remaining attributes; rows $\{\phi(w_i)\}_{i\neq j}$ match $\phi(w_j)$ over subsets of J1 and have distinct symbols for the remaining attributes.

Consider the mapping σ' which is obtained from σ by replacing $\sigma(w_j)$ by $\phi(w_j)$. This replacement will not affect any other row in σ, as a consequence of the matching characteristics of simple template dependencies. Thus, σ' differs from σ only in the symbols corresponding to $\Pi_{NotJ1}\sigma'(w_j)$. There is a homomorphism h that maps σ into σ':

$$\forall\ k\neq j,\ \sigma'(w_k) = h(\sigma(w_k)) = \sigma(w_k)$$
$$\Pi_{J1}\sigma'(w_j) = \Pi_{J1}h(\sigma(w_j)) = \Pi_{J1}\sigma(w_j)$$
$$\Pi_{NotJ1}\sigma'(w_j) = \Pi_{NotJ1}h(\sigma(w_j)) = \Pi_{NotJ1}\phi(w_j)$$

This corresponds to the containment mapping $\sigma'\subseteq\sigma$. An inverse homomorphism h^{-1} can be derived in a similar way and results in $\sigma\subseteq\sigma'$. Therefore, $\sigma\equiv\sigma'$, which means that $\phi(w_j)$ is equivalent to $\sigma(w_j)$. Thus $\phi(w_j)$ is subsumed by $\overline{w}_{1d}$, contradicting the assumption in the previous page that this is not true for any j.

Thus, the set $\{t\}_i \cup \overline{w}_{1d}$ is sufficient to maintain the constraints $\{C_{i_n}\}$, and perform the request $(De,\{t\}_i,R_i,exception)$. ■

Lemma 2.2: (minimality while maintaining consistency)
No subset of the deletions described by $De(\{t\}_i,R_i,force)$ can perform the corresponding underlying operation without violating the constraints $\{C_{i_n}\}$.
Proof

In the absence of constraints, or in case of functional dependencies, the proof is trivial, since $De(\{t\}_i,R_i,force) : \Delta_i = [\{t\}_i]$. The proof that follows is for template dependencies, where $De(\{t\}_i,R_i,force) : \Delta_i = [\{t\}_i \cup \overline{w}_{1d}]$.

Deletion of $\{t\}_i$ is certainly necessary. Let $\{t_{rem}\}$ be defined as in the previous lemma. Assume, by contradiction, that the set indicated is not minimal. Consider the symbolic valuation as a logic sentence

$$\overline{w}_1 \wedge \cdots \wedge \overline{w}_s \Rightarrow \{t\}_i \cup \{t_{rem}\}.$$

When $\{t\}_i$ is deleted, the constraint can only be maintained if the left side of the sentence is made false or $\{t_{rem}\} \neq \emptyset$. Assume $\{t_{rem}\} = \emptyset$, and thus $\overline{w}_{1d}$ should be considered.

If $\{t\}_i \cup \overline{w}_{1d}$ is not minimal, then there must exist a subset of the tuples described by $\overline{w}_{1d}$ that achieve the deletion and maintain the consistency. Let this subset be denoted by $\overline{w}_{dp}$. Assume $\overline{w}_{1d}$ corresponds to some substituted row $\overline{w}_j$. There exists at least one tuple t_j such that t_j is described by $\overline{w}_{1d}$ and t_j is not deleted when $\overline{w}_{dp}$ is deleted. This means that there may exist a set of tuples $\{t_{k,k\neq j}\}$ described by $\{\overline{w}_{k,k\neq j}\}$ such that $\bigwedge_{k=1}^{s} (t_k \in r_i) \Rightarrow t_{s+1} \in r_i$. In other words, after deletion of the set of tuples described by $\overline{w}_{dp}$, t_{s+1} must still exist to maintain consistency. This, however, corresponds to a state where $\overline{w}_1 \wedge \cdots \wedge \overline{w}_s \Rightarrow \{t\}_i$ holds. Consequently, not all tuples described by $\{t\}_i$ can be eliminated if only a subset of $\overline{w}_{1d}$ is deleted.

Therefore, the rows generated by $De(\{t\}_i, R_i, force)$ constitute a minimal set. ■

Theorem 3: The set of rows generated by $In(\{t\}_i, R_i, force)$ describes a minimal set of insertions and deletions necessary to accomplish the corresponding underlying operation and maintain the semantic integrity of r_i as defined by the constraints $\{C_{i_n}\}$.

In the absence of constraints the proof is trivial, since $In(\{t\}_i, R_i, force)$: $\Delta_i = \{t\}_i$, and thus the proof refers to cases where $\{C_{i_n}\} \neq \emptyset$.

This Theorem is proved by separately proving sufficiency and minimality of the set generated by $In(\{t\}_i, R_i, force)$; minimality of deletions must also be considered in the case of maintenance of functional dependencies. The proof considers the possible compositions of $\{C_{i_n}\}$. Lemmas 3.1 and 3.2 prove sufficiency and minimality when $\{C_{i_n}\}$ consists of functional dependencies only; Lemmas 3.3 and 3.4 prove sufficiency and minimality when $\{C_{i_n}\}$ consists of a simple template dependency; and Lemmas 3.5 through 3.10 refer to the correctness of $In(\{t\}_i, R_i, force)$ when $\{C_{i_n}\}$ contains both functional dependencies and a simple template dependency. In particular, Lemmas 3.5 through 3.8 prove no ambiguous updates are introduced by limiting interaction between functional dependencies and a simple template dependency to classes (HXA), (HX) and (A), and Lemmas 3.9 and 3.10 prove sufficiency and minimality of the set generated by $In(\{t\}_i, R_i, force)$ for these classes.

As a consequence of the definition of necessarily redundant insertions with respect to partial dependencies, given at the beginning of this chapter, only full dependencies need to be considered when dealing with minimality and sufficiency of the set of insertions generated by $In(\{t\}_i, R_i, force)$. The interaction between the non-embedded part of a partial template dependency and a set of functional dependencies can also be disregarded. The procedure for determining the value of system-generated parametric variables, described in Chapter 3, ensures that functional dependencies affecting the non-embedded part are never violated by the additional insertions $\{(\overline{w}_{s+1})_j\}$. For this reason, these insertions need not be checked for violation of such functional dependencies. This characteristic of template dependencies is also recognized in a related context: that of testing implication of partial template dependencies [MAI83, Theorem 14.9]. (An X-partial template dependency TD is implied by a set of template dependencies when the chase with T-rules results in a row that matches the conclusion of TD over X.) Thus, unless specified, all template dependencies mentioned in the proof of Theorem 3 are assumed to be full dependencies.

Lemma 3.1: (sufficiency for functional dependencies) Let $\{C_{i_n}\}$ consist of a set of functional dependencies in Partitionable BCNF. The set of insertions and deletions generated by $In(\{t\}_i,R_i,force)$ accomplishes the corresponding underlying operation, and does not violate the constraints $\{C_{i_n}\}$ for relation r_i.
Proof

For functional dependencies, $In(\{t\}_i,R_i,force)$: $\Delta_i = \{t\}_i \cup \{(w_{fd},[w'_{fd}])\} = \{t\}_i \cup \Delta I(t_{FD}) - \Delta D(t_{FD})$. Proving sufficiency requires proving that these operations accomplish insertion of $\{t\}_i$ and do not require further insertions or deletions to maintain the constraints $\{C_{i_n}\}$. The fact that $\{t\}_i$ is inserted is obvious: all expressions in $\Delta D(t_{FD})$ disagree with $\{t\}_i$ over at least one attribute, and therefore cannot delete $\{t\}_i$. Thus, only maintenance of consistency needs to be proved.

Let $\bar{r}_i = r_i \cup \{t\}_i \cup \Delta I(t_{FD}) - \Delta D(t_{FD})$, and $t_{FD} \in \Delta I(t_{FD})$. Assume, initially, that R_i is in BCNF. Furthermore, assume, by contradiction, that there is some functional dependency X→A violated in $\bar{r}_i$, i.e.,

$$\exists\ t_1,t_2 \in \bar{r}_i \mid \Pi_X t_1 = \Pi_X t_2 \wedge \Pi_A t_1 \neq \Pi_A t_2.$$

Since the relation before the update was consistent and deletions do not violate functional dependencies, then the violation must have been introduced by some insertion described by $\{t\}_i \cup \Delta I(t_{FD})$. Assume that insertion of some tuple t_1 is responsible for this violation.

The following cases are possible:

○ t_1 is described by $\{t\}_i$
in this case, a pair $(w_{fd},[w'_{fd}])$ is generated such that
$(\Pi_X w_{fd} = \Pi_X w'_{fd} = \Pi_X t_1) \wedge (\Pi_A w_{fd} = \Pi_A t_1) \wedge (\Pi_A w_{fd} \neq \Pi_A w'_{fd})$
This means that w'_{fd} describes all tuples $t_2 \in r_i$, where $(\Pi_X t_2 = \Pi_X t_1) \wedge (\Pi_A t_1 \neq \Pi_A t_2)$. Therefore, since w'_{fd} describes a set of deleted tuples, t_2 is deleted, and the dependency is not violated.

○ t_1 is described by $\Delta I(t_{FD})$
this means that $\exists\ t_{FD} \mid t_1 = t_{FD}$. Since $\Pi_{XA} t_{FD} = \Pi_{XA} w_{fd} = \Pi_{XA}\{t\}_i$, then a violation exists if

$$\exists t_2 \mid \Pi_X t_2 = \Pi_X w'_{fd} \wedge \Pi_A t_2 = \Pi_A w'_{fd},$$

and the same reasoning is used as before.

Because the relation is in BCNF, no insertion w_{fd} requires in its turn that further changes be forced. For any two functional dependencies {X→A, Y→B} $\in \{C_{i_n}\}^+$, $X^+ = Y^+$. Thus, $\Pi_{X^+} w_{fd} = \Pi_{X^+}\{t\}_i$, and insertion of any tuple described by $\{w_{fd}\}$ does not violate dependency which are not already violated by insertion of $\{t\}_i$.

Assume now that R_i is not in BCNF, but rather in Partitionable BCNF. An insertion generates sets of changes $(w_{fd},[w'_{fd}])$ for each BCNF partition, and these changes do not violate the functional dependencies in the partition because they are sufficient for a BCNF relation. Since there is no dependency connection among partitions, all attributes in $(w_{fd},[w'_{fd}])$ that do not belong to the partition are represented by placeholder variables, and thus the changes in a partition cannot affect another partition. The changes for each partition can be processed independently and the order in which they are processed does not affect the final result.

Therefore, the set of insertions and deletions generated by $In(\{t\}_i, R_i, force)$ is sufficient to maintain the consistency when there are only functional dependencies in $\{C_{i_n}\}$. ■

Lemma 3.2: (minimality while maintaining consistency) Let $\{C_{i_n}\}$ be composed of a set of functional dependencies. No subset of the insertions and deletions generated by $In(\{t\}_i, R_i, force)$ can perform the corresponding underlying operation without either violating the constraints $\{C_{i_n}\}$ or destroying additional information.

Proof

Assume that for a given functional dependency X→A the set of insertions and deletions $(w_{fd}, [w'_{fd}])$ generated to maintain this dependency is not minimal. In other words, there exists a subset of the operations described, $(E_{fd}, [E'_{fd}])$, that is sufficient to maintain the consistency and preserve the information in r_i. This means that $(E_{fd}, [E'_{fd}])$ describe a set of insertions and deletions of all tuples (t,[t']) such that

$$t' \in r_i \wedge \Pi_X t' = \Pi_X \{t\}_i \wedge \Pi_A t' \neq \Pi_A \{t\}_i \text{ and}$$
$$\Pi_{XA} t = \Pi_{XA} \{t\}_i .$$

If $(E_{fd}, [E'_{fd}])$ is a subset of $\{(w_{fd}, [w'_{fd}])\}$, then there exists at least one pair of inserted and deleted tuples, (u,[u']) that is described by $\{(w_{fd}, [w'_{fd}])\}$ and is not described by $(E_{fd}, [E'_{fd}])$. In other words,

$$\Pi_{XA} u = \Pi_{XA} w_{fd} \wedge \Pi_{XA} u' = \Pi_{XA} w'_{fd} .$$

However, this means that

$$\Pi_X u' = \Pi_X \{t\}_i \wedge \Pi_A u' \neq \Pi_A \{t\}_i .$$

Therefore (u') must also be deleted, and thus if $[E'_{fd}]$ does not include u', it is not sufficient to maintain the constraint.

If (u') is deleted, then (u) must be inserted, otherwise information will be lost. Even though (E_{fd}) describes a set of insertions which is sufficient to maintain consistency, it does not preserve information in (R_i - XA). Therefore, the set of insertions and deletions $\{(w_{fd}, [w'_{fd}])\}$ is a minimal set that maintains the consistency of all functional dependencies without destroying more information than necessary. ■

Lemma 3.3: (sufficiency for a simple template dependency) Let $\{C_{i_n}\}$ consist of a simple template dependency. The set of insertions generated by $In(\{t\}_i, R_i, force)$ accomplishes the insertion and does not violate the constraints $\{C_{i_n}\}$ for relation r_i.

Proof

Assume by contradiction that the set $\{t\}_i \cup \{(\overline{w}_{s+1})_j\}$ is not sufficient to maintain the consistency of a template dependency. This means that there exists some additional insertion expression I_0 which must be considered to maintain the consistency.

The set $\{(\overline{w}_{s+1})_j\}$ describes all possible additional insertions required by the insertion of $\{t\}_i$, and therefore I_0 is not an insertion expression needed to allow insertion of $\{t\}_i$. Thus, I_0 must be needed in order to allow some insertion in the set $\{(\overline{w}_{s+1})_j\}$. In other words, there exists some expression $E \in \{(\overline{w}_{s+1})_j\}$ such that E must be inserted to allow insertion of $\{t\}_i$, and if E is inserted, then I_0 must also be inserted, otherwise the dependency will be violated. Without loss of generality, assume E and I_0 are generated by the mappings

$$\sigma\text{: } \sigma(w_k) \leftarrow \{t\}_i \wedge \sigma(w_{s+1}) = \text{E}$$
$$\phi\text{: } \phi(w_j) \leftarrow \text{E} \wedge \phi(w_{s+1}) = I_0.$$

where σ is generated by substitution in the original template dependency TD, and ϕ is generated by substitution in a copy of the dependency, TD$'$, whose symbols do not appear in TD. Since the proof is based on row matching for simple dependencies, this also includes the case where more than one row is replaced by $\{t\}_i$ (or E).

Since only simple (full) dependencies are considered, the following holds:
: I_0 has the same symbols as E for all attributes in $\mathcal{S}$ame(j,s+1);
: I_0 has new symbols (from TD$'$) for the remaining attributes, that are assigned by rows that do not match w_j.
: $\forall\ w_i, w_j$, $\mathcal{S}$ame(i,j) $\subseteq$ $\mathcal{S}$ame(i,s+1) and $\mathcal{S}$ame(i,j) $\subseteq$ $\mathcal{S}$ame(j,s+1).
Therefore, the only symbols of E that will propagate to other rows of ϕ correspond to those in $\mathcal{S}$ame(j,s+1), since E replaces row w_j. All the other symbols in ϕ will be new.

Consider the mapping ϕ' obtained from ϕ by replacing $\phi(w_j)$=E by $\sigma(w_j)$, yielding the conclusion I'_0. Since $\sigma(w_j)$ comes from the original template substitution which yielded E, then E has the *same* symbols as $\sigma(w_j)$ for attributes in $\mathcal{S}$ame(j,s+1). Thus, replacing $\phi(w_j)$=E by $\sigma(w_j)$ will not cause any changes to the other rows of mapping ϕ. Thus, $I'_0 \equiv I_0$ is the expression that must be inserted to maintain consistency if ϕ' holds, i.e., if $\sigma(w_j) \in r_i$ and $\bigwedge_{l \neq j} (\phi(w_l) \in r_i)$.

If $j \neq k$, $\{\sigma(w_j)\}$ and $\{\phi(w_i)_{i \neq j}\}$ describe tuples that already existed in the relation before the update, or tuples in $\{t\}_i$. Therefore, since the relation was originally consistent, I_0 must also have existed before the update, or correspond to some other insertion expression in $\{(\overline{w}_{s+1})_j\}$, and cannot describe a new set of insertions. If $j = k$, then $\sigma(w_j) = \{t\}_i$, and I_0 is equivalent to E (i.e., it corresponds to an insertion which has already been described in mapping σ). Thus, I_0 cannot represent a new insertion.

Since no insertion is needed beyond those generated by $In(\{t\}_i, R_i, force)$, this set is sufficient to accomplish the underlying update request while maintaining the consistency when $\{C_{i_n}\}$ consists of a simple template dependency. ■

Lemma 3.4: (minimality while maintaining consistency) Let $\{C_{i_n}\}$ consist of a simple template dependency. No subset of the insertions generated by $In(\{t\}_i, R_i, force)$ can perform the required insertion without violating the template dependency.
Proof

Assume that there exists a set of insertions, $\{I_0\}$, which is contained in the set Δ_i generated by $In(\{t\}_i, R_i, force)$, and that these insertions maintain the database consistency and achieve the desired insertion. In other words, there is at least one tuple t' described by Δ_i that is not described by $\{I_0\}$. This means that $\{I_0\}$ is a subset of $\{(\overline{w}_{s+1})_j\}$ which is sufficient to ensure consistency after insertion of $\{t\}_i$.

Let the consistent relation before the insertion be represented by a tableau, and add $\{t\}_i$ to this tableau. If this new tableau is to represent the consistent relation state after the insertion, then it must be chased with the template dependency to derive which additional tuples must be inserted. This is done by combining the existing rows with the new rows in all possible mappings. (This process corresponds to deriving the relation state description obtained when $\{t\}_i$ replaces each template hypothesis row and substitution rules are applied.) By the definition of a tableau chase using T-rules (see Chapter 2), the additional tuples obtained exactly correspond to all valuations of the set described by $\{(\overline{w}_{s+1})_j\}$. Therefore, no subset of $\{(\overline{w}_{s+1})_j\}$ is sufficient to maintain the consistency, since this means the chase with T-rules yields superfluous expressions which are not needed for maintaining the relation consistency.

Thus, the set of insertions described by Δ_i constitute a minimal set which maintains the relation consistency when $\{C_{i_n}\}$ consists of a simple template dependency. ■

Lemmas 3.5 through 3.10 refer to relation schemes that are subject to both functional dependencies and a template dependency. Recall from Chapter 3 that the syntactic constraints imposed on the database scheme R are that every relation scheme must be in Partitionable BCNF and that
a) if there exists at most one non-trivial partition, its interaction with the template dependency must be one of the following
. class (A) or
. class (HXA) or
. class (HX);
b) if there exists more than one non-trivial partition, only class (HXA) interactions are allowed.

The lemmas that follow justify this choice of allowed interactions.

Lemma 3.5: Consider the scheme R<$\{A_{i_k}\}$,$\{C_{i_n}\}$>, where $\{C_{i_n}\}$ contains a simple full template dependency and a set of non-trivial functional dependencies in Partitionable BCNF. Let the dependency X→A be a full functional dependency in $\{C_{i_n}\}$. Let $\{(\overline{w}_{s+1})_j\}$ denote insertions required to maintain consistency with respect to the template dependency when a single tuple t denoted by $\{t\}_i$ is inserted into an instance r_i such that $r_i \cup \{t\}_i - \Delta D(t_{FD})$ satisfies X→A. If for all j,

a) $\forall$ k, $\Pi_A w_k = \Pi_A w_{s+1}$ (class A) or

b) $\exists$ k, $\Pi_{XA} w_k = \Pi_{XA} w_{s+1}$ (class HXA)
then the tuples described by any one $(\overline{w}_{s+1})_j$ necessarily satisfy X→A.

Proof

a) Let all rows match over A. All mappings generated by replacement of a row by an expression in $\{t\}_i \cup \Delta I(t_{FD})$ create insertion expressions $\{(\overline{w}_{s+1})_j\}$ where $\Pi_A(\overline{w}_{s+1})_j = \Pi_A\{t\}_i$ (or $\Pi_A(\overline{w}_{s+1})_j = \Pi_A t_{FD}$) and therefore no two tuples described by any such expression can disagree over XA.

b) Let both right hand side and left hand side come from a single row w_k. If the insertion expression $(\overline{w}_{s+1})_k$ is generated by replacing w_k by $\{t\}_i$ (or t_{FD}), then all tuples it describes have their XA-values taken from $\{t\}_i \cup \Delta I(t_{FD}) - \Delta D(t_{FD})$. If $(\overline{w}_{s+1})_j$ is generated by replacing some other row w_j by $\{t\}_i$ (t_{FD}), the tuples it describes have their XA-values taken from tuples in $r_i \cup \{t\}_i \cup \Delta I(t_{FD})$ - $\Delta D(t_{FD})$, which agree over XA. In neither case can the tuples described by any insertion expression disagree over XA. ■

Lemma 3.6: If a template dependency interacts with a functional dependency according to class (HXA), then the set of additional insertions $\{(\overline{w}_{s+1})_j\}$ does not require any deletions in order to maintain the relation consistency with respect to this functional dependency.

Proof

Let $\bar{r}_i = r_i - \Delta D(t_{FD})$. Consider a partition containing X→A. Let $\bar{x}$ $\bar{a}$, the symbols for XA in the conclusion row, appear in some template row w_k. Two kinds of insertion expression can exist as far as the attributes XA are concerned:

a) $\Pi_{XA}(\overline{w}_{s+1})_k = \Pi_{XA}\{t\}_i$, when $\{t\}_i$ replaces w_k (or $\Pi_{XA}(\overline{w}_{s+1})_k = \Pi_{XA} t_{FD}$, when t_{FD} replaces w_k).

b) $\Pi_{XA}(\overline{w}_{s+1})_j = \Pi_{XA}\overline{w}_k$, when $\{t\}_i$ (or t_{FD}) replaces some other row, and $\overline{w}_k$ describes a set of tuples which belong to $\bar{r}_i \cup \{t\}_i \cup \Delta I(t_{FD})$.

The tuples described by $\bar{r}_i \cup \{t\}_i \cup \Delta I(t_{FD})$ satisfy X→A (Lemma 3.1). Thus, any expression type (a) agrees with $\bar{r}_i \cup \{t\}_i \cup \Delta I(t_{FD})$, and no two expressions of type (a) can disagree over XA. Expressions type (b) cannot disagree with $\bar{r}_i \cup \{t\}_i \cup \Delta I(t_{FD})$ because the XA-values are taken from this very same set of tuples. For the same reason, expressions of type (b) cannot disagree with each other over XA, and expressions of type (b) cannot disagree with any expression type (a). Thus, when the interaction between a template dependency and a functional dependency is of class (HXA), the insertions in $\{(\overline{w}_{s+1})_j\}$ can be processed as if this functional dependency did not exist, and no further deletions are necessary to maintain this dependency. ■

Corollary to Lemma 3.6: If all interactions in a partition are of class (HXA), then the insertions in $\{(\overline{w}_{s+1})_j\}$ need not consider any functional dependency in this partition for the effect of additional deletions. If for all partitions the interactions are of class (HXA), $\{(\overline{w}_{s+1})_j\} = \{(\overline{w}_{s+1})_j^G\}$.

Lemma 3.7: Assume $\{C_{i_n}\}$ contains more than one non-trivial BCNF partition interacting with a template dependency. If at least one interaction is of class (A), the updates generated by $In(\{t\}_i, R_i, force)$ depend on the order in which the tuples in $\{t\}_i \cup \Delta I(t_{FD})$ are pushed down the template.

Proof

Let $\bar{r}_i = r_i - \Delta D(t_{FD})$. Let one partition contain X→A, and consider $t_{FD_1} \in \{t\}_i \cup \Delta I(t_{FD})$. Because the interaction is class (A), all insertion expressions that result from pushing t_{FD_1} down the template obey $\Pi_A(\overline{w}_{s+1})_j = \Pi_A t_{FD_1}$, and cannot disagree with each other over XA. Thus, the only violations of X→A that need to be considered correspond to checking $\{(\overline{w}_{s+1})_j\}$ against $\bar{r}_i \cup \{t\}_i \cup \Delta I(t_{FD})$ - t_{FD_1}.

Assume that $t_{FD_1} \in \Delta I(t_{FD})$ and that insertion of t_{FD_1} requires deletion of some expression in $\Delta I(t_{FD})$, say t_{FD_2}. The set of tuples that will be inserted, ΔI_1, is

$$\Delta I_1 \subseteq \{t\}_i \cup (\Delta I(t_{FD}) - t_{FD_2}) \cup \{(\overline{w}_s+1)_j{}^G\}.$$

Suppose now that t_{FD_2} is pushed down the template ahead of t_{FD_1}, and that the resulting insertions require deletion of some tuple t_{FD_3}. Thus, if t_{FD_2} is processed before t_{FD_1}, the set of tuples that is inserted by $In(\{t\}_i, R_i, force)$ is

$$\Delta I_2 \subseteq \{t\}_i \cup (\Delta I(t_{FD}) - t_{FD_2} - t_{FD_3}) \cup \{(\overline{w}_s+1)_j{}^G\}.$$

In particular, if $t_{FD_3} = t_{FD_1}$, then ΔI_2 does not delete t_{FD_2} and thus $\Delta I_1 \neq \Delta I_2$, and the contents of the set generated by $In(\{t\}_i, R_i, force)$ depend on the order in which the insertions are checked. ■

Example: Let X =(MN). Consider the following template, and the dependencies MN→A, Y→B. The partitions are (MNA), (YB). The template interacts with the first partition according to class (A), and with the second according to class (HXA).

M	N	A	Y	B
m_1	k_1	a	y_1	b_1
k_2	n_1	a	y_2	b_2
m_1	n_1	a	y_1	b_1

Forced insertion of (MNAYB) causes the changes $(w_{fd},[w'_{fd}])$

$(MNAy_1b_1, [MNA'y_1b_1])$ and $(m_1n_1aYB, [m_1n_1aYB'])$.

Let $t_{FD_1} = MNAyb$ be a tuple described by $(MNAy_1b_1)$, and $t_{FD_2} = M\beta A''YB$ be a tuple described by (m_1n_1aYB). Assume that before the insertion r_i also contained some tuple $(\alpha\beta A\gamma\delta)$. Pushing t_{FD_1} down the template generates

(Mn_1Ayb) and $(m_1NAy_1b_1)$,

when the first and the second template rows are replaced. Considering $(\alpha\beta A\gamma\delta)$, valid valuations for these expressions are $u_1 = (M\beta Ayb)$ and $u_2 = (\alpha NA\gamma\delta)$. Tuple u_1 disagrees with t_{FD_2} over (MNA), and thus to maintain consistency either insertion of t_{FD_2} or insertion of u_1 should be prevented.

If insertion of t_{FD_2} is prevented, then it will not be pushed down the template and the final updated relation will not contain the tuples that result from this operation. If, instead, insertion of u_1 is prevented, then by applying deletion rules to the mapping that created u_1, the first hypothesis row containing a parametric variable $(\overline{w}_{1d})$ must be eliminated, i.e. t_{FD_1} will not be inserted. In the latter case, the insertions originating from pushing t_{FD_2} down the template will be performed.

Lemma 3.8: Consider $\{C_{i_n}\}$ consisting of a template dependency and functional dependencies such that there is only one non-trivial BCNF partition and let $\bar{r}_i = r_i - \Delta D(t_{FD})$. Let the dependencies in the partition interact with the template according to classes (A), (HX) or (HXA). If $\{(\overline{w}_s+1)_j{}^B\} \neq \emptyset$, all tuples in $\{(\overline{w}_s+1)_j{}^B\}$ originate from $\bar{r}_i$.

Proof

This requires showing that all insertion expressions agree with each other. Consequently, if they are to disagree with any expression, this expression must describe tuples in $\bar{r}_i$. In other words, no insertion requires deletion of some other insertion.

The attributes can be assigned to two partitions, of which one (possibly empty) is not subject to any functional dependency. Let X→A ∈ $\{C_{i_n}\}$ and $\Pi_{XA} w_{s+1} = \bar{x}\ \bar{a}$. Let W = R_i-(X ∪ A). Forced insertion of (XAW) creates the changes

(XAW,[XA'W]) and (XAw,[XA'w]), i.e., $\Delta I(t_{FD})$ = (XAw).

Thus, $\{(\bar{w}_{s+1})_j\}$ results from pushing (XAW) and (XAw) down the template.

If the interaction is of class (HXA) with respect to XA, then by Lemma 3.6 no insertion expression requires any further deletions, and $\{(\bar{w}_{s+1})_j\}$ = $\{(\bar{w}_{s+1})_j^{\ G}\}$ (and $\{(\bar{w}_{s+1})_j^{\ B}\} = \emptyset$).

Assume now it is of class (HX), i.e., there exists at least one row that contains $\bar{x}$ and no row containing $\bar{x}\ \bar{a}$. There are four types of insertion expression $(\bar{w}_{s+1})_j$ that can be generated when some row is replaced by $\{t\}_i$ or by t_{FD} ∈ $\Delta I(t_{FD})$

: when only the row(s) containing $\bar{x}$ are replaced, $\Pi_{XA}(\bar{w}_{s+1})_j$= (X$\bar{a}$);
: when only the row(s) containing $\bar{a}$ are replaced, $\Pi_{XA}(\bar{w}_{s+1})_j$ = ($\bar{x}$A);
: when row(s) containing $\bar{a}$ and rows containing $\bar{x}$ are replaced, $\Pi_{XA}(\bar{w}_{s+1})_j$ = XA;
: when row(s) containing neither $\bar{a}$ nor $\bar{x}$ are replaced, $\Pi_{XA}(\bar{w}_{s+1})_j = \bar{x}\bar{a}$.

In this last situation, the XA-values are taken from tuples that agree with $\bar{r}_i$ ∪ $\{t\}_i$ ∪ $\Delta I(t_{FD})$ over XA, and no deletions are needed. Thus, the only tuples that may need to be deleted to prevent inconsistency are those whose A-value is different from 'A'. Since there exists only one nontrivial partition, $\Pi_{XA}\Delta I(t_{FD}) = XA$, and therefore the tuples that need to be deleted do not belong to $\{t\}_i$ ∪ $\Delta I(t_{FD})$. Consequently, only tuples from $\bar{r}_i$ may need to be deleted.

Class (A) means this template has no row containing $\bar{x}$, since this would create an interaction of class (HXA). If it is class (A) with respect to XA, then
: $\Pi_{XA}(\bar{w}_{s+1})_j = \dot{x}A$, where $\dot{x}$ is not completely parametric, or
: $\Pi_{XA}(\bar{w}_{s+1})_j = XA$.
Similarly, only the tuples in $\bar{r}_i$ need to be considered for possible additional deletions.

This proof can be extended to more attributes on the right hand side, and to the other dependencies in the partition. In particular, class (A) with respect to some dependency X→A means it is class (A) in $\{C_{i_n}\}$, otherwise the symbol for X would appear in some hypothesis row, which would make the interaction class (HXA). Thus, for a single non-trivial BCNF partition and a template dependency that interact according to classes (A), (HX) or (HXA), only tuples from $\bar{r}_i$ need to be deleted. (Notice this does not occur when there exists more than one non-trivial BCNF partition, because then $\Pi_{XA}\Delta I(t_{FD})$ is not necessarily equal to $\Pi_{XA}\{t\}_i$ — see previous lemma.) ■

Lemma 3.9: (sufficiency of the set generated by $In(\{t\}_i, R_i, force)$) The rules $In(\{t\}_i, R_i, force)$ for generating Δ_i when $\{C_{i_n}\}$ contains functional dependencies and a template dependency interacting within the allowed classes define a set of insertions and deletions for r_i which are sufficient to satisfy the dependencies in $\{C_{i_n}\}$.

Proof

Let $\Delta D = \Delta D(t_{FD}) \cup \{(\overline{w}_s+1)_j{}^B\}$, and $\Delta I = \{t\}_i \cup \Delta I(t_{FD}) \cup \{(\overline{w}_s+1)_j{}^G\}$.

a) Sufficiency with respect to the set of functional dependencies

Assume the updates in Δ_i are not sufficient to maintain some dependency X→A. Then, there exist at least two tuples t_1, t_2 described by

$$r_i \cup \{t\}_i \cup \Delta I(t_{FD}) \cup \{(\overline{w}_s+1)_j{}^G\} - (\Delta D(t_{FD}) \cup \{(\overline{w}_s+1)_j{}^B\})$$

that violate X→A. Since deletions do not violate functional dependencies, and r_i was consistent before the insertion, then this violation must have been introduced by the set ΔI. Two cases are possible:

a) $t_1 \in r_i$ and t_2 is described by ΔI;

b) both t_1 and t_2 are described by ΔI.

Assume case (a) can occur. This means that there exists a tuple t_2 described by ΔI, and some tuple t_1 in r_i such that $\Pi_X t_2 = \Pi_X t_1 \wedge \Pi_A t_2 \neq \Pi_A t_1$. However, this means that not all tuples described in ΔI are checked against r_i, which is not true. Thus, case (a) cannot occur.

Case (b) may occur in the following combinations:

b.1) $t_1 = \{t\}_i$ and $t_2 \in \Delta I(t_{FD})$

b.2) t_1 is described by $(\{t\}_i \cup \Delta I(t_{FD}))$ and t_2 is described by some expression in $\{(\overline{w}_s+1)_j{}^G\}$;

b.3) t_1 and t_2 are described by different insertion expressions $(\overline{w}_s+1)_j, (\overline{w}_s+1)_k \in \{(\overline{w}_s+1)_j{}^G\}$;

b.4) t_1 and t_2 are described by the same insertion expression $(\overline{w}_s+1)_j \in \{(\overline{w}_s+1)_j{}^G\}$.

- (b.1) cannot occur by Lemma 3.1
- (b.2) cannot occur for more than one partition, since in this case only class (HXA) is allowed. For one partition only, for any functional dependency X→A $\in \{C_{i_n}\}$, $\Pi_{XA}\{t\}_i = \Pi_{XA}\Delta I(t_{FD})$, and each insertion expression $(\overline{w}_s+1)_j$ is checked against $\{t\}_i \cup \Delta I(t_{FD})$. Any tuple t_1 described by $(\overline{w}_s+1)_j$ that does not agree with $\{t\}_i$ has its insertion prevented by some deletion in $\{(\overline{w}_s+1)_j{}^B\}$ (i.e., if $\bigwedge_j \sigma(\overline{w}_j) \Rightarrow t_1$, then $\bigwedge_{j\neq k} \sigma(\overline{w}_k) \wedge \neg \overline{w}_i \not\Rightarrow t_1$). Thus, for every t_1 corresponding to a valuation of $(\overline{w}_s+1)_j$ that disagrees with $\{t\}_i \cup \Delta I(t_{FD})$, there exists some deletion in ΔD that precludes the need for the existence of t_1.
- (b.3) cannot occur because for the templates analyzed all insertion expressions agree with each other over XA.
- (b.4) cannot occur by Lemma 3.5 for classes (HXA) and (A). For class (HX), since there exists a single non-trivial partition, the disagreeing expressions are those that disagree with $\{t\}_i$ over XA. Their insertion is prevented, and thus even if case (b.4) occurs, it is circumvented by deletions in Δ_i.

Thus, the insertions specified in Δ_i do not violate any functional dependency. □

b) Sufficiency with respect to the template dependency

Assume the operations are not sufficient to maintain the relation's consistency with respect to the template dependency. This can occur either if there exists some tuple insertion or some tuple deletion that is omitted.

An insertion is omitted if either a) one of the row substitutions is not performed, or b) an insertion in ΔI is cancelled by a deletion in ΔD. Case (a) does not occur: Assume case (b) can occur, i.e., an insertion I_0 that must be performed in order to obtain a consistent final state is prevented by a deletion D_0 in ΔD (i.e., $I_0 \equiv D_0$). Any insertion in ΔI is performed only when the associated relation state holds. Any tuple D_0 in ΔD is pushed up the template, and thus is effectively deleted (Theorem 2). This also eliminates one of the conditions that would require the insertion of I_0 in order to maintain the relation consistency. Thus, $I_0 \equiv D_0$ means that I_0 is not necessary for a consistent final state, and consequently the set of insertions ΔI is sufficient *given the set of deletions* ΔD.

A deletion is omitted if there exists some tuple that should be deleted and was not, i.e., a) the expression that describes this tuple was not processed against the template or b) some deleted tuple is re-inserted by some expression in ΔI. Case (a) does not occur: Assume further deletions may need to be performed. This means some deletion in $\Delta D(t_{FD}) \cup \{(\overline{w}_{s+1})_j{}^B\}$ needs to be pushed up the template. However, since this is a simple template dependency, one iteration is sufficient to produce the deletions that will maintain consistency (Theorem 2).

Case (b) cannot occur: Tuples in $\Delta D(t_{FD})$ cannot be re-inserted by tuples in $\{t\}_i \cup \Delta I(t_{FD})$ (Lemma 3.1). Thus, a necessary deletion D_0 in ΔD is re-inserted by some necessary insertion I_0 iff it is inserted by $\{(\overline{w}_{s+1})_j{}^G\}$. If I_0 is necessary, there exists a set of tuples that created the need for this insertion in order to obey the template dependency. In other words, the deletion expression that describes D_0 has not been processed against the template (since one of the conditions that would make insertion of D_0 necessary is deleted through deletion of some $\overline{w}_{1d}$). Since this cannot occur, no necessary deletion is re-inserted by some necessary insertion in ΔI, and the set of deletions is sufficient. □

c) Sufficiency with respect to the functional dependencies and the template dependency

No insertion in ΔI will re-insert a tuple deleted by ΔD, and *vice versa*. The insertions are sufficient to maintain both the template dependency and the functional dependencies. The deletions are sufficient to maintain consistency with respect to the template dependency, and cannot violate any functional dependency. Therefore, the insertions and deletions in Δ_i are sufficient to obtain a consistent final state. ■

Lemma 3.10: (minimality of the set generated by $In(\{t\}_i, R_i, force)$) The rules $In(\{t\}_i, R_i, force)$ for generating Δ_i when $\{C_{i_n}\}$ contains functional dependencies {F} and a template dependency that interact in the allowed classes define a minimal set of insertions and deletions for r_i which satisfy the dependencies in $\{C_{i_n}\}$.

Proof

Assume first the set of deletions is not minimal. This set is divided into two components: tuples that have to be deleted in order to maintain the consistency with respect to the functional dependencies (say, $\{t_\alpha\}$); and tuples that have to be deleted to maintain consistency with respect to the template dependency (say, $\{t_\beta\}$). Given $\{t_\alpha\}$, then $\{t_\beta\}$ is a minimal set of deletions which maintains the consistency and allow the deletions in $\{t_\alpha\}$ to occur (Theorem 2). Thus, the set of deletions is not minimal if and only if the set $\{t_\alpha\}$ is not minimal.

Assume that there exists a subset of $\{t_\alpha\}$ that can maintain the relation's consistency with respect to {F}, given the insertions $\{t\}_i \cup \Delta I(t_{FD}) \cup \{(\overline{w}_s{+}1)_j\}$. The deletions in $\{t_\alpha\}$ are of three types: the ones determined in the preliminary functional dependency processing, the ones that prevent insertion of additional tuples that will cause violation of X→A, and tuples in $r_i - \Delta D(t_{FD})$ that violate X→A given the insertions in $\{(\overline{w}_s{+}1)_j{}^G\}$. If any such deletion is omitted, then by construction there may exist an inserted tuple which causes the updated relation to violate the functional dependency. Thus, the set of deletions $\{t_\alpha\}$ is minimal.

Assume now the set of insertions is not minimal. Again, this set can be divided into: insertions needed in order to maintain the functional dependencies ($\Delta I(t_{FD})$, which is a minimal set that preserves information *given the deletions in* $\Delta D(t_{FD})$ — Lemma 3.2); and insertions needed to maintain the template dependency ($\{(\overline{w}_s{+}1)_j{}^G\}$, already proved to be minimal by Lemmas 3.3 and 3.4). Thus, although strictly speaking none of the tuples in $\Delta I(t_{FD})$ *needs* to be inserted, omitting any would unnecessarily destroy information. Thus, the set of insertions is a minimal set.

Assume now that even though both sets are minimal, it is possible to eliminate some insertions and deletions from Δ_i such that the result will still be consistent. The set of deletions $\Delta D(t_{FD})$ is necessary to maintain the functional dependencies (Lemma 3.2); the accompanying set $\Delta I(t_{FD})$ prevents loss of information. Thus, if the set Δ_i is not minimal, the set of insertions and deletions that can be eliminated from it must come from $\{(\overline{w}_s{+}1)_j{}^G \cup (\overline{w}_s{+}1)_j{}^B\}$.

Assume there exist some inserted tuples $\{t_{in}\}$ described by $\{(\overline{w}_s{+}1)_j{}^G\}$ and some deleted tuples $\{t_{de}\}$ described by $\{(\overline{w}_s{+}1)_j{}^B\}$ such that $\nabla = r_i \cup \Delta_i \cup \{t_{de}\} - \{t_{in}\}$ is consistent. As shown in part (b) of the proof of this lemma, the set $\{(\overline{w}_s{+}1)_j{}^G\}$ is necessary to maintain the consistency of the template dependency, given that $\{t\}_i \cup \Delta I(t_{FD})$ are inserted. Thus, ∇ is consistent only if $\{t_{in}\} \subseteq \{t_{de}\}$. However, as also shown in this proof, no deletion in ΔD deletes an insertion in ΔI, and *vice-versa*. This means that $\{t_{in}\} \cap \{t_{de}\} = \emptyset$, and therefore $\{t_{in}\} = \emptyset$.

Thus, $\{t_{de}\}$ must be such that $r_i \cup \Delta_i \cup \{t_{de}\}$ is consistent, which has been shown by this lemma to be false, since the set of deletions is necessary.

Therefore, the set of insertions and deletions in Δ_i constitute a minimal set of updates that maintain consistency and preserve information. ■

Theorem 4: The set of insertions and deletions generated by $Ch(\{t,t'\}_i, R_i, force)$ describes a minimal set of symbolic row operations to accomplish the corresponding underlying operation and maintain the semantic integrity of r_i as defined by the constraints $\{C_{i_n}\}$.

Lemma 4.1: (sufficiency while maintaining consistency) The set of insertions and deletions described by $Ch(\{t,t'\}_i, R_i, force)$ accomplishes the corresponding underlying operation and does not violate the constraints $\{C_{i_n}\}$ of relation r_i.

Proof

a) For no constraints, the proof is trivial.

b) For functional dependencies only, the same proof as for Lemmas 3.1 and 3.2 for insertions.

c) For template dependencies only, the proof is a combination of the proof of

Lemma 2.1 for $De(\{t'\}_i, R_i, \mathit{force})$ followed by the proof of Lemmas 3.3 and 3.4 for $In(\{t\}_i, R_i, \mathit{force})$ and $In(\overline{w}_{1c}, R_i, \mathit{force})$.

d) For functional dependencies and a template dependency, Lemmas 3.9 and 3.10 prove the correctness of $In(\{t\}_i, R_i, \mathit{force}) \cup In(\overline{w}_{1c}, R_i, \mathit{force})$. ■

Lemma 4.2: (minimality while maintaining consistency) No subset of the insertions and deletions described by $Ch(\{t,t'\}_i, R_i, \mathit{force})$ can perform the corresponding underlying operation without either violating the constraints $\{C_{i_n}\}$ or unnecessarily destroying information.

Proof

$In(\{t\}_i, R_i, \mathit{force}) : \Delta_i'$; $In(\overline{w}_{1c}, R_i, \mathit{force}) : \Delta_i''$;
$Ch(\{t,t'\}_i, R_i, \mathit{force}) : \Delta_i = (\overline{w}_{1c}, [\overline{w}'_{1c}]) \cup (\{t\}_i, [\{t'\}_i]) \cup \Delta_i' \cup \Delta_i''$. The proof again follows the previous proofs:

a) For absence of constraints, it is trivial;

b) For functional dependencies alone, it is the same as Lemmas 3.1 and 3.2.

c) For template dependencies alone, it is the same as minimality for forced deletion of $\{t'\}_i$, followed by minimality of forced insertion of $\{t\}_i \cup \overline{w}_{1c}$, where $\overline{w}_{1c}$ is inserted to preserve information.

d) Template dependencies and functional dependencies are handled as follows:

- forcing $(\{t\}_i, [\{t'\}_i])$ requires deletion of $\{t'\}_i$, which is minimally achieved by deleting $\overline{w}'_{1c}$ (Lemma 2.2).

- forcing $(\{t\}_i, [\{t'\}_i])$ requires insertion of $\{t\}_i$, which is minimally achieved by the operations in $In(\{t\}_i, R_i, \mathit{force})$ (Lemma 3.10).

- $\overline{w}_{1c}$ must be inserted if the change operation is to preserve information, and this insertion is minimally achieved by $(In, \overline{w}_{1c}, R_i, \mathit{force})$ (Lemma 3.10).

Therefore, the set generated by $Ch(\{t,t'\}_i, R_i, \mathit{force})$ constitutes a minimal set of insertions and deletions that performs the change, maintains the consistency, and preserves information. ■

Theorem 5: The set of rules $State(Op_i)$, that determine the underlying conditions for a row to appear in each initial tableau T_1, exactly describe the relation state for which the corresponding set of tuples exist.

The proof consists of stating the rules described in chapter 3 as logic clauses. These rules correspond to defining the conditions under which an update should be performed. Recall that the symbolic valuation of a template row $(\overline{w}_j)$ can also be interpreted as $True(\overline{w}_j)$. For this proof it means that the set of tuples described by $\overline{w}_j$ exist in $r_i \cup \{t\}_i \cup \Delta I(t_{FD}) - \Delta D(t_{FD})$. This Theorem is proved separately for conditional and forced updates.

Lemma 5.1: (Rules for conditional updates)

The set of rules in $State(Op_i)$ that determine the conditions for execution of the underlying operation $(Op, \{t\}_i, R_i, \text{cond})$ exactly describe the class of relation states in which the indicated update may occur without violating the constraints $\{C_{i_n}\}$.

Proof

Insertions

1. $\{C_{i_n}\}$ contains a functional dependency X→A
   $\{t\}_i$ is inserted if $\forall\ t \in r_i,\ (\Pi_X t = \Pi_X \{t\}_i) \Rightarrow \Pi_A t = \Pi_A \{t\}_i$.
2. $\{C_{i_n}\}$ contains a template dependency

Let $(\overline{w}_1 \cdots \overline{w}_{j-1}\ \{t\}_i\ \overline{w}_{j+1} \cdots \overline{w}_s)|(\overline{w}_{s+1})_j$ correspond to the substituted template when $\{t\}_i$ replaces w_j. (Recall that since $\{t\}_i$ plays multiple roles, other expressions $\overline{w}_l$ may also be equivalent to $\{t\}_i$.)

$\{t\}_i$ is inserted if $\bigwedge_j [\bigwedge_{k=1}^{s} (\overline{w}_k) \Rightarrow (\overline{w}_{s+1})_j]$

Deletions

1. $\{C_{i_n}\}$ contains an X-partial template dependency

Let $(\overline{w}_1 \cdots \overline{w}_s)|\{t\}_i$ be the replaced template.

$\{t\}_i$ is deleted if $(\ \overline{w} \in \bar{r}_i \mid \Pi_X\overline{w} = \Pi_X\{t\}_i) \vee (\nexists\ j|\ \overline{w}_j \subseteq \{t\}_i \Rightarrow \bigvee_{k=1}^{s} (\neg\overline{w}_k))$.

The first part of this clause results directly from the properties of partial template dependencies.

Changes

1. $\{C_{i_n}\}$ contains a functional dependency X→A

$\{t'\}_i$ is changed into $\{t\}_i$ if

$\{t'\}_i \in r_i \wedge (\exists\ t \in r_i|\ \Pi_X t = \Pi_X\{t\}_i) \Rightarrow \Pi_A t = \Pi_A\{t\}_i$

2. $\{C_{i_n}\}$ contains a template dependency

Let the modified template be $\{(\overline{w}_j,[\overline{w}'_j]\} \mid (\{t\}_i,[\{t'\}_i])$

$\{t'\}_i$ is changed into $\{t\}_i$ if $\{t'\}_i \in r_i \wedge [(\exists\ j|\ \overline{w}_j \neq \overline{w}'_j) \Rightarrow \neg\overline{w}'_j]$

3. $\{C_{i_n}\}$ contains both functional dependencies and a template dependency

Same as (2.) above, and $(\overline{w}_j \cup \{t\}_i)$ agree over the functional dependencies.

Lemma 5.2: (Rules for forced updates)

The set of rules in *State*(Op_i) that determine the underlying conditions for additional updates to be represented in the initial tableau T_1, given the the underlying operation $(Op_i,\{t\}_i,R_i,\text{force})$, exactly describe the class of relation states in which the indicated update *must* occur in order to maintain the constraints $\{C_{i_n}\}$.

Proof

Insertions

1. $\{C_{i_n}\}$ contains a functional dependency X→A

perform $(w_{fd},[w'_{fd}])$, where $\{w_{fd}\} = \Delta I(t_{FD})$ and $\Pi_{XA} w_{fd} = \Pi_{XA}\{t\}_i$, if

$\exists\ t'|\ (\Pi_X t' = \Pi_X\{t\}_i\ \wedge\ \Pi_A t' \neq \Pi_A\{t\}_i)$, where $\{t'\}$ is described by $\{w'_{fd}\}$.

2. $\{C_{i_n}\}$ contains a template dependency

Let $(\overline{w}_1 \cdots \overline{w}_{j-1}\ \overline{w}_a\ \overline{w}_{j+1} \cdots \overline{w}_s)|(\overline{w}_{s+1})_j$ correspond to the substituted template when $\overline{w}_a$, an insertion expression, replaces w_j.

$(\overline{w}_{s+1})_j$ is inserted if $[\bigwedge_{k=1}^{s} (\ \overline{w}_k)] \wedge \neg (\overline{w}_{s+1})_j$

3. $\{C_{i_n}\}$ contains a functional dependency X→A and a template dependency

○ I $\in (\overline{w}_{s+1})_j^G$ is inserted if

$[\bigwedge_{k=1}^{s} (\ \overline{w}_k)]\ \wedge\ \neg\ (\overline{w}_{s+1})_j\ \wedge\ (\nexists$ t described by I $\mid \Pi_X t = \Pi_X\overline{w}_a\ \wedge\ \Pi_A t \neq \Pi_A\overline{w}_a)$.

○ D $\in (\overline{w}_{s+1})_j^B$ is deleted if

D describes tuples in r_i and $\exists$ I $\in (\overline{w}_{s+1})_j^G$ | D disagrees with l over XA
$\vee$
D corresponds to additional deletions $\overline{w}_{1d}$, determined by the rules for deletions that follow.

Deletions

1. $\{C_{i_n}\}$ contains a (X-partial) template dependency
Let $(\overline{w}_1 \cdots \overline{w}_{1d} \cdots \overline{w}_s) | \{t\}_i$ be the replaced template.
$\overline{w}_{1d}$ is deleted if

$$(\nexists\ \overline{w} \in r_i - \{t\}_i \mid \Pi_X \overline{w} = \Pi_X \{t\}_i) \wedge (\nexists\ j | \ \overline{w}_j \subseteq \{t\}_i) \wedge [\bigwedge_{j=1}^{s} (\overline{w}_j)]$$

Changes

1. $\{C_{i_n}\}$ contains a functional dependency X→A
the same as for forced insertions.
2. $\{C_{i_n}\}$ contains a template dependency
Let the modified template be $\{(\overline{w}_l, [\overline{w'}_l]\} \mid (\{t\}_i, [\{t'\}_i])$
the additional change $(\overline{w}_{1c}, [\overline{w}'_{1c}])$ is performed if $\bigwedge_{j=1}^{s} (\overline{w}'_j)$
The remaining conditions (for insertions of $\overline{w}_{1c}$ and $\{t\}_i$) are the same as the ones for forced insertions.
3. $\{C_{i_n}\}$ contains a template dependency and functional dependencies
The same as for the two previous cases. $\overline{w}_{1c}$ must agree with $\{t\}_i$ over XA. ■

Theorem 6: The update validation algorithm always stops.
Proof

The algorithm stops with an error message if the input is invalid. The algorithm also stops if no error occurs. Step 1 (checking input consistency and building the templates) is finite, being limited by the size of the input and the number of rows in each template. Building the BCNF partitions is finite, since it involves checking a finite number of functional dependencies. Creating the underlying modification table entries is a finite process for simple templates and Partitionable BCNF.

The process of building each tableau sequence is finite. Building T_1 involves one scan of the (finite) underlying modification table. Processing the remainder of each sequence is also finite, and the proof is similar to the proof of finiteness of traditional tableau chases: it consists of simulating a (finite) number of monotonic operations, computed by the rules $Ex(q_v)$, followed by functional dependency checking. This process can only change placeholder variables either to other placeholder variables with lower index, or placeholder to parametric variables (but never the opposite). Furthermore, it can only change system-generated parametric variables either to system-generated parametric variables with lower index or to user-entered parametric variables. This means that, in any sequence $T_1, T_2 \ldots T_k$, no tableau is repeated, i.e., $\nexists$ l,j | l $\neq$ j and $T_j = T_l$, and the number of placeholder variables is a non-increasing non-negative quantity. The number of sequences is finite, being limited by the number of relation schemes in the database. Therefore, the algorithm always stops. ■

Theorem 7: The contribution to the SJP view V=(R,q_v) of the set of underlying tuples described in a tableau T_1 is exactly described by the final tableau T_k of the corresponding tableau sequence generated by the update validation algorithm.

This Theorem is proved using four Lemmas. Lemma 7.1 proves that the $Ex(q_v)$ execution rules, which provide the transition from tableau T_j to tableau T_{j+1} in a sequence $T_1 \cdots T_k$, correctly process symbolic tuple expressions according to the operations specified in q_v. Lemma 7.2 shows that checking functional dependencies at each step does not invalidate the results represented in the rows of tableau T_{j+1} after execution of $Ex(q_v)$. Lemma 7.3 shows that the rows of tableau T_k describe at least all possible isolated contributions to the view of the underlying tuples in T_1. Lemma 7.4 shows that the rows of T_k do not describe any update that cannot occur in the view.

Lemma 7.1: The execution rules $Ex(q_v)$, that determine how to perform relational operators on tableau rows, correctly represent the effect of the select, project and join operations specified in q_v for the extensions r_i of R_i.
Proof

The execution rules $Ex(q_v)$ may introduce errors either in processing tableau rows or in processing the entries of the database state table. Each case will be analyzed separately.

Case 1: Processing of tableau rows

In the tableau rows, errors can happen either because the $Ex(q_v)$ rules introduce invalid symbols, or because the actual operations are not simulated correctly. $Ex(q_v)$ rules simulate the actual operations correctly, because they represent a modified version of computation of tableau queries. The $Ex(q_v)$ rule for projection corresponds to the projection rule for computation of tableau queries (see [MAI83, Chapter 10] and Chapter 2 of this thesis). The $Ex(q_v)$ rule for joins corresponds to the rule for computing joins in tableau queries (after variables are unified according to the chase J-rule). Tableau queries handle selection with equality by making the selected attribute have a specific value. In the $Ex(q_v)$ rules this has been modified by introducing parametric variables that describe the range of the selected tuples, which are thereafter treated as in tableau query processing. Since actual operations are simulated correctly, errors can only occur if incorrect symbolic values are assigned to variables when performing the simulation.

The initial set of rows in T_1 is correct (see Theorems 1 through 4). Consider a transition from tableau T_j to T'_{j+1}, achieved through the execution of some rule in $Ex(q_v)$, and assume T_j contains no error. (T_{j+1} will be obtained from T'_{j+1} by checking functional dependencies.) No row of T_j is ignored by $Ex(q_v)$: all rows that describe tuples that should be processed by the relational operation are processed, and the remaining rows are copied into T'_{j+1}. Thus, errors can only be introduced by the modified rows.

Projection rules do not introduce new variables in the tableau.

Selection rules only modify the variables corresponding to attributes in the selection range and either a) change a placeholder to a parametric value without modifying any other characteristics of the tableau variables or b) if the attribute subject to the select condition is already represented by some parametric variable A_1, the variable is maintained, and the description of the new selection condition is appended to the description of A_1. No other variables are modified and the net result is that of imposing a (further) restriction on the tuples being processed.

The rules for joining $R_1 \times R_2$ can only introduce errors while determining the final value of the unified attributes in $R_1 \cap R_2$. These rules will be analyzed for each possible case of joining pairs of attributes:

1. (placeholder, placeholder): the rule is the same as the one used in processing tableau queries with joins, and cannot therefore introduce errors.
2. (placeholder, parametric): parametric variables stand for specific values. Since the user (or the system) indicates that a given value (or range of values) will be used for that attribute at run time, the only tuples that are needed for determining *modifications* in the view are those where that value is matched. Therefore, parametric variables should determine the outcome of the join.
3. (parametric, parametric): if at least one of the variables is user-entered, the result must match this variable (for the same reasons as above). If both values are system-generated, then again the rule is the same as that of processing tableau queries.

Thus, Case 1 cannot introduce errors: the operational specification of q_v is correct.

Case 2: Processing database state table entries

The initial entries in the database state table are correct, as proved by Theorem 5. Assume therefore that some error is introduced by the $Ex(q_v)$ rules during a transition from T_j to T'_{j+1}. The database state table entries are modified either when variables have their values changed or when two rows are joined.

Every time a tableau variable has its value changed, the change is correct (see Case 1), and therefore must be reflected in the database state table entries. Two tuples (i.e. rows) can only be joined if they exist at the same time (which is reflected in the database state table by having the entries for both rows combined into a conjunctive expression). Thus, the entries in the database state table are correctly processed.

Summarizing, the operational specification of $Ex(q_v)$ is correct, it does not assign invalid values to variables, and all necessary conditions for the execution of a given operation are stored in the database state table. Therefore, the execution rules are correct. ∎

Lemma 7.2: Let T'_{j+1} denote a tableau that results from applying $Ex(q_v)$ to tableau T_j. The check of functional dependencies over the rows of T'_{j+1}, which results in the next tableau T_{j+1} in the sequence $T_1 \cdots T_k$, does not invalidate the results described in T'_{j+1}.

Proof

Recall that, for each row of T'_{j+1}, functional dependency checking assigns parametric variables to attributes in the right side of a functional dependency whenever the left side has been assigned a specific (i.e., non-selection) parametric variable during execution of join $Ex(q_v)$ rules. Furthermore, dependencies are excluded from this process if

- they apply to relation schemes not yet processed by $Ex(q_v)$; or

- they have already assigned parametric values to attributes on their right side; or
- the attributes corresponding to their right side are eliminated by projection.

Any change in a variable of a row in T'_{j+1} is reflected in the database state table. Thus, if functional dependency checking does not invalidate the rows of T'_{j+1}, neither does it introduce errors in this table. The proof shows that functional dependency checking does not introduce errors, by proving that:
- it does not create inconsistencies among the rows of T'_{j+1}
- it needs not check a different set of functional dependencies
- it does not interfere with the set of valid valuations of the rows of T'_{j+1} because

: it does not create additional expressions (i.e., that describe updates that can never occur) and

: it does not eliminate valid expressions (i.e., does not discard the description of tuples that should have been considered as resulting from the underlying updates).

In other words, it will be proved that checking functional dependencies cannot invalidate any expression in T'_{j+1}, neither can it violate underlying constraints.

Assume, first, that checking functional dependencies invalidates the expressions in T'_{j+1} by assigning parametric values to some variables, thereby creating an inconsistency among the rows of T_{j+1}. This cannot occur because each tableau row corresponds to an independent update and is processed separately. Notice this process cannot create expressions whose valuations violate any functional dependency, since all parametric values assigned correspond to system-determined variables.

Assume, now, that functional dependency checking should consider a different set of functional dependencies. Obviously, it needs not consider the dependencies that apply to relations not yet processed by $Ex(q_v)$. If the left side of a given dependency does not consist of (specific values) parametric variables, there is no point in checking the dependency, either. Therefore, the only dependencies that need to be checked are those whose left side consists of parametric variables. Once these dependencies are checked, and the symbols on the right side are changed, they need not be checked again: any future joins will preserve the parametric variables assigned, and the corresponding functional dependencies cannot determine additional parametric variables. Finally, if all attributes corresponding to the right side of a dependency are discarded by projection, their value will not affect the view formation, and the corresponding dependency needs not be checked. Thus, the set of dependencies being considered at each step is correct.

Assume now that checking functional dependencies modifies some expression E in T'_{j+1} so as to make it describe a larger set of tuples than it originally did. Let E' denote this expression after this modification. Since functional dependency checking assigns new parametric values to variables, E' subsumes E. Therefore, E' cannot describe *more* updates than E: it describes *less* tuples than E.

Assume, finally, that since E' describes less tuples, the functional dependency checking discarded the description of view tuples which may correspond to a view update. This means, however, that there exists some functional dependency X→Y and a view tuple t such that if $\Pi_X t$ is a specific value, $\Pi_Y t$ is not necessarily the corresponding value. This obviously contradicts the notion of a functional dependency. Therefore, no valid valuation is discarded by restricting E to E', since all "real world" tuples which are described by E are also described by E'.

Thus, the results described by the rows of T'_{j+1} are maintained by T_{j+1}. ■

Lemma 7.3: Any SJP view tuple which can be formed by applying q_v to the expressions displayed in T_1 is described by some row of the final tableau T_k.

Proof

Assume, by contradiction, that there exists some view tuple r_v which is affected (i.e., inserted, deleted or changed) by the isolated updates described in T_1, and yet is not described by any row in T_k. If this tuple is affected by the updates in T_1, then it must originate from a set of tuples $\{t_1 \cdots t_j\}$ described by expressions $\{\rho_1 \cdots \rho_l\}$ in T_1.

Since r_v belongs to the view, it is generated by applying the view generating function q_v to the tuples $\{t_1 \cdots t_j\}$, i.e., $r_v = q_v(t_1 \cdots t_j)$. In symbolic terms, r_v is described by $q_v(\rho_1 \cdots \rho_l)$. By Lemma 7.1, the execution rules $Ex(q_v)$ correctly perform the operations specified in q_v and no tuples are ignored in the process. Thus, $q_v(\rho_1 \cdots \rho_l)$ is correctly performed by applying $Ex(q_v)$ to $\{\rho_1 \cdots \rho_l\}$. This means $Ex(q_v)(\rho_1 \cdots \rho_l)$ *must* be represented by a row in T_k - i.e., r_v is described by some row in T_k. Therefore, there exists no view update that is based on the underlying tuples described in T_1 and which is not described by some row in T_k. ■

Lemma 7.4: Any result row that appears in the final tableau T_k correctly describes a possible update on an SJP view resulting from the isolated updates described in the tableau T_1.

Proof

Assume, by contradiction, that there is a superfluous tuple described by some row in T_k, i.e., a tuple that cannot originate from the underlying operations described in T_1. This means that, in generating the sequence $T_1 \cdots T_k$, a transition from tableau T_j to tableau T_{j+1} creates an expression that describes tuples which did not originate from the tuples described in T_1. Since any tableau row is obtained by a sequence of operations q_v on rows of T_1, introduction of additional tuples can only occur if there is an error in the execution rules $Ex(q_v)$ or in functional dependency checking, which is impossible (see Lemmas 7.1 and 7.2). Therefore, no row of T_k can represent the isolated contribution to the view of tuples which are not described by the expressions in T_1. ■

Theorem 8: The database state table entries associated with each output row in a final tableau T_k in the update validation algorithm exactly describe the class of database states necessary for the corresponding update to be reflected in the view.

Proof

Assume, by contradiction, that an entry in the database state table for some result row contains an incomplete description of the database state necessary for the corresponding update to appear in the view (i.e., further conditions than stated are needed). The individual underlying conditions necessary for generation of tableau T_1 are correctly derived by $State(Op_i)$ substitution rules (Theorem 5). Let T_j and T_{j+1} be successive tableaux in the sequence $T_1 \cdots T_k$. The transition from T_j to T_{j+1} is achieved by executing one operator from q_v using $Ex(q_v)$ (Lemma 7.1) followed by checking functional dependencies (Lemma 7.2); any change in attribute values is recorded, indicating what are the conditions for this operator to be executed. If more conditions were needed, the operator could not be executed, and the tableau T_{j+1} would not be created. Therefore, the database state table entries generated are sufficient for a) initially obtaining the underlying update in T_1 and b) reflecting it on the view as described by T_k. Thus, the database state description associated with the output rows is sufficient.

These conditions also constitute a necessary set: the conditions in $State(Op_i)$ are necessary (Theorem 5). If any intermediate database state computation were eliminated, the function q_v could not be executed to the end, and the output row would not be generated. Thus, the database states associated with each row are necessary and sufficient for describing the class of underlying states for which the update occurs. ■

Theorem 9: For monotonic view functions, deleting a tuple t from an underlying relation extension r_i does not add tuples to the view extension; adding t to r_i does not delete tuples from the view extension.
Proof

Deleting t_i from r_i creates $r'_i = r_i - t_i$; inserting t_i in r_i creates $r'_i = r_i \cup t_i$. For deletions from r_i, no tuples are added to the view $q_v(r_1 \ldots r_n)$, since, by the monotonicity of q_v, $q_v(r_1 \ldots r_{i-1}\ r'_i\ r_{i+1} \ldots r_n) \subseteq q_v(r_1 \ldots r_n)$.
For insertions in r_i, no tuples are deleted from the view, since $q_v(r_1 \ldots r_n) \subseteq q_v(r_1 \ldots r_{i-1}\ r'_i\ r_{i+1} \ldots r_n)$.

Thus, if the result of an isolated underlying update to r_i is the deletion(insertion) of some tuple t_v in the view, then no other underlying deletion(insertion) in some r_j, $j \neq i$, will re-insert(delete) t_v. ■

Theorem 10: Any possible outcome for a complete update policy $u_v = \{(Op_i, \{t\}_i, R_i, exception)\}_{i=1..n}$ on an SJP view is derivable from the set described by the rows of $\{T_k\}_{i=1..n}$ representing the isolated contributions of the underlying updates $u_i = (Op_i, \{t\}_i, R_i, exception)$ to the same view.
Proof

The proof is based on the fact that the update validation algorithm makes sure that underlying operations do not interact destructively. Lemmas 10.1 and 10.2 show that the description of an isolated update is complete, and that the order in which the relations are updated does not affect the final view. Lemma 10.3 shows that the description of any update which may appear on the view can be derived by combining the results of isolated updates from STEP 2.

Lemma 10.1: Given an initial database extension described by $\{r_1 \cdots r_n\}$, an SJP view formed by $q_v(R_1 \cdots R_n)$, and a set of underlying updates $\{u_i = (Op_i, \{t\}_i, R_i, exception)\}$, the order in which the relations are updated does not affect the final database extension $\{r'_1 \cdots r'_n\}$, nor the final view extension $q_v(r'_1 \cdots r'_n)$.
Proof

Assume the final database state can be influenced by the order in which the operations are executed. This means that some update to relation r_i causes changes to $r_j \neq r_i$. This is not possible in the model studied, for SJP views, which assumes independence of relation schemes.

Therefore, the final database state does not depend on the order in which underlying updates are executed. Since the final view extension is defined as a function over the final database state, the order of the underlying operations does not affect the final view state. ■

Lemma 10.2 (non-interference) Given a set of updates $u_v = \{u_i = (Op_i, \{t\}_i, R_i, exception)\}$ and an SJP view, if the set of all possible effects on the view caused by one underlying operation u_i is given by the set of expressions W_i, then there exist database states for which the set of view modifications described by W_i will occur, independent of the underlying updates to the other relations $\{r_{k \neq i}\}$.
Proof

a) **Non-interference among deletions**

Let u_1=(De,$\{t\}_1$,R_1,exception), and u_2=(De,$\{t\}_2$,R_2,exception) represent two underlying deletions, where the effects of u_1 and u_2 on the view are given by W_1 and W_2.

As a corollary to Theorem 9, deletions cannot be undone by other deletions. Thus, no deletion expressed in W_1 can prevent a deletion expressed in W_2, and *vice versa,* even though W_1 and W_2 may have expressions in common. This means that some deletions expressed in W_1 as depending on the existence of $\{t\}_2$ in r_2 are the same deletions described in W_2 as depending on the existence of $\{t\}_1$ in r_1. Thus, if u_1 is executed before u_2, some of the deletions expressed in W_2 will occur as part of W_1, and *vice versa.*

b) **Non-interference among insertions**

Let u_1=(In,$\{t\}_1$,R_1,exception), and u_2=(In,$\{t\}_2$,R_2,exception) represent two underlying insertions, where the effects of u_1 and u_2 on the view are given by W_1 and W_2, and neither u_1 nor u_2 force functional dependencies.

Again by Theorem 9, insertions cannot be undone by other insertions. Since there are no functional dependencies involved, execution of u_1 and u_2 implies that no deletions will be performed in r_1 and r_2. Thus, no insertion described in W_1 can be prevented by an insertion in W_2, and *vice versa,* even though W_1 and W_2 may have expressions in common. This means that if u_1 is executed before u_2, some of the tuples described by W_2 will occur as part of W_1, and vice-versa.

c) **Non-interference among insertions and deletions**

Let u_1= (In,$\{t\}_1$,R_1,exception), u_2= (De,$\{t\}_2$,R_2,exception), and u_3 =(Ch,$\{t,t'\}_3$,R_3,exception) represent underlying updates, where the insertion u_1 may force functional dependencies. Let W_1,W_2 and W_3 represent the isolated effects of u_1,u_2 and u_3 on the view.

W_1 may contain the expressions E_1, $[E'_1]$ and $(E_1,[E'_1])$;
W_2 may contains expressions $[E_2]$;
W_3 may contain the expressions E_3, $[E'_3]$ and $(E_3,[E'_3])$.
Since there is no interference among insertions, the expressions in E_1 and E_3 do

not depend on the order in which the insertions in r_1 and r_3 are executed, nor in their content. Similarly, deletions E'_1,E_2 and E'_3 do not depend on the order in which the deletions in r_1,r_2 and r_3 are executed.

Assume that some deletion described by $[E_2]$ can be prevented by some insertion in r_1, i.e., deletion of $[E_2]$ depends on the existence of tuples $\{t\}_1$ in r_1 which will not exist after insertions. If $\{t\}_1$ does not exist after insertion in r_1, this means that $\{t\}_1$ contributes to forming the expressions $[E'_1]$. Therefore, the corresponding deletion will always occur, being described both by $[E_2]$ and $[E'_1]$.

Insertions described by E_1 cannot be prevented by the deletions described by $[E'_1]$, because they are executed for a single relation r_1 over disjoint sets of tuples. The same applies to the expressions E_3 and $[E'_3]$.

Finally, assume by contradiction that some insertion described in E_1 or E_3 will never occur because it depends on the existence of tuples that have been deleted by u_1,u_2 or u_3. This means, for instance, that some E_1 corresponds to the statement

S1: view tuples Γ_v are inserted if $\bar{t} \in r_2$,

whereas some E_2 corresponds to the statement

S2: view tuples Γ_v are deleted because $\bar{t} \in r_2$ is deleted.

View insertion S1 cannot obviously occur, since S2 indicates u_2 deletes $\bar{t}$. Recall that when an initial tableau T_1 is built from the underlying modification table, the only tuples from r_2 that are allowed to participate in view insertions are those that will *not* be deleted by any operation in r_2. Thus, the only tuples from r_2 allowed to participate in a view insertion have the form $t \neq \bar{t}$. Therefore, no expression E_1 can correspond to statement S1. By the same token, no insertion described by E_3 can be prevented by deletions in u_1 or u_2. Thus, the expressions in W_1,W_2 and W_3 describe view updates which may eventually occur, and combinations of underlying updates do not prevent any such expression from being reflected on the view. ■

Lemma 10.3 Any possible effect of a complete policy $u_v = \{(Op_i, \{t\}_i, R_i, exception)\}$ on an SJP view can be derived by combining the information contained in the sets W_i that describe the isolated outcomes of underlying updates $(Op_i, \{t\}_i, R_i, exception)$.

Proof

From Lemma 10.2, the isolated contribution W_i to an SJP view caused by one update $u_i = (Op_i, \{t\}_i, R_i, exception)$ is completely described by the output of the update validation algorithm, and is not affected by other simultaneous underlying operations. Assume, by contradiction, that there exists a view update which results from a set of underlying updates $\{u_i = (Op_i, \{t\}_i, R_i, exception)\}$ and yet cannot be derived from a valid valuation of the set $\bigcup_i W_i$. Consider the following facts

- the final effect of a set of updates does not depend on the order in which sets of updates are executed, as long as the complete policy is treated as a transaction (Lemma 10.1);
- a given relation can only contribute to modifying an SJP view if the relation is modified itself (by the definition of SJP views); and
- the net effect of a sequence of underlying updates is that of eliminating some

tuples and adding others to the view.

Let r_v be a view tuple that is affected by the complete policy, and assume its description is not derivable from $\bigcup_i W_i$. If r_v is a result of the complete policy, then it is a result of applying the updates in u_v in some order, i.e., there is a set of underlying updates $u_1 \cdots u_j$ that contribute to r_v. Assume, without loss of generality, that at least u_1 contributes to the formation of r_v. Since W_1 describes all possible effects on the view of the policy u_1, then the description of r_v must be contained in W_1, otherwise u_1 would not contribute to forming r_v. Therefore, the only possibility for r_v not being derivable from the set $\bigcup_i W_i$ is that some underlying update(s) u_k whose effect is described by W_k, interact destructively with u_1, in such a way that some of the updates described in W_1 will not occur as a consequence. This, however, cannot happen, as shown by Lemma 10.2. Therefore, any possible effect of a complete policy u_v on an SJP view can be derived from a valid valuation of the set $\bigcup_i W_i$ that describes the isolated outcome of underlying updates $u_i = (Op_i, \{t\}_i, R_i, exception)$. ■

4.3. Complexity analysis

Theorem II:

The number of comparison operations executed by the update validation algorithm is polynomial in the size of the scheme R=$\{R_i < \{A_{i_j}\}, \{C_{i_n}\} > \}$.
Proof

Let Nrel be the number of relations in R, Nh_i be the number of hypothesis rows in R_i, Na_i the number of attributes in the template for R_i, and NF_i the number of non-redundant functional dependencies in scheme R_i.

The complexity of the operations in STEP 1 (input consistency checking) depends on the number of schemes with functional dependencies. Consider one scheme R_i. Building BCNF partitions requires a sweep of all functional dependencies in the relation, as well as checking the closure of each determinant in a partition. This takes $O(Na_i \times NF_i)$ [MAI83, chapter 4] per determinant, and thus $O(Na_i \times NF_i^2)$ per relation. Checking for valid interactions between functional dependencies and template dependencies takes $O(NF_i \times Nh_i \times Na_i)$ comparisons. Let $NA = \max_i(Na_i)$, $NF = \max_i(NF_i)$, and $NH = \max_i(Nh_i)$.

Thus, STEP 1 takes $O(Nrel \times NA \times \max(NF^2, NF \times NH))$.

The creation of the underlying modification table in STEP 2 requires execution of $Op(\{t\}_i, R_i, exception)$ and derivation of the rules $State(Op_i)$. In the expressions that follow, all rows formed by pairs of expressions (e.g., $(w_{fd}, [w'_{fd}])$) will be counted as two entries in a table. Each execution of $De(\{t\}_i, R_i, exception)$ takes $Nh_i \times Na_i$ comparisons when $\{C_{i_n}\}$ contains a template dependency. Execution of $In(\{t\}_i, R_i, exception)$ and $Ch(\{t, t'\}_i, R_i, exception)$ varies according to the type of constraint involved. Assume the worst case for changes, when every attribute to be changed is involved in some functional dependency.

If $\{C_{i_n}\}$ consists of functional dependencies only, then both insertions and changes take approximately $Na_i \times NF_i$ comparisons, to create the pairs $\{(w_{fd},[w'_{fd}])\}$. If $\{C_{i_n}\}$ consists of one simple template dependency, the number of comparisons for $In(\{t\}_i, R_i, cond)$ is $Nh_i \times (Nh_i - 1) \times Na_i$ (since the substitution rules must be applied as many times as there are rows in the template); and execution of $Ch(\{t,t'\}_i, R_i, cond)$ is composed of at worst $3 \times Nh_i \times Na_i$ comparisons for finding w_{1c} (since each row is duplicated and the elements of each pair must be compared against each other); forced changes need an additional $2 \times Nh_i \times (Nh_i - 1) \times Na_i$ comparisons for determining additional insertions (resulting from $\{t\}_i$ and $\overline{w}_{1c}$).

If $\{C_{i_n}\}$ consists of both functional dependencies and one template dependency, $In(\{t\}_i, R_i, cond)$ takes approximately $Na_i \times (NF_i - Nh_i + Nh_i^2)$ comparisons; $In(\{t\}_i, R_i, force)$ may require an additional $Na_i \times NF_i \times (Nh_i^2 - Nh_i)$ comparisons for pushing $\Delta I(t_{FD})$ down the template, and $(Nh_i \times Na_i)(Nh_i + NF_i + NF_i . Nh_i)$ comparisons to push potential deletions up the template. $Ch(\{t,t'\}_i, R_i, cond)$ takes approximately $Na_i \times (NF_i + 2Nh_i + Nh_i^2)$ comparisons; $Ch(\{t,t'\}_i, R_i, force)$ may require $Na_i \times (2Nh_i^2 + 2Nh_i)$ more comparisons than $In(\{t\}_i, R_i, force)$, to account for additional insertions and deletions.

Building the tableau T_1 for each underlying update, and initializing the database state table entries, takes constant time. The remainder of STEP 2 depends on the number of joins and of checking functional dependencies.

Let NJ denote the number of joins, and NT the number of rows in the first tableau. Recall that, for each tableau sequence needed to process an update to scheme R_i, T_1 contains one row for each scheme $R_{j \neq i}$. The number of rows for R_i is the number of expressions in $Op(\{t\}_i, R_i, exception)$. If each relation may contain both functional dependencies and a template dependency, the following results hold.

. for conditional insertions and deletions $\begin{cases} NT = Nrel \\ NJ = Nrel - 1 \end{cases}$

. for conditional changes $\begin{cases} NT = Nrel \\ NJ \leq 2 \times (Nrel - 1) \end{cases}$

. for forced deletions $\begin{cases} NT \leq Nrel + 1 \\ NJ \leq 2 \times (Nrel - 1) \end{cases}$

. for forced insertions $\begin{cases} NT \leq (Nrel + Nh_i \times NF_i + 5Nh_i + 2NF_i) \\ NJ \leq (1 + 5Nh_i + 3NF_i + NF_i \times Nh_i) \times (Nrel - 1) \end{cases}$

This expression for NT is composed as follows: NF_i rows $(w_{fd},[w'_{fd}])$; $(Nh_i)(NF_i + 1)$ insertions for pushing $\{t\}_i \cup \Delta I(t_{FD})$ down the template; this may generate a potential $2Nh_i$ deletions, which have to be pushed up the template together with $\{w'_{fd}\}$, resulting in a total $4Nh_i + NF_i$ deletion expressions.

. for forced changes, $\begin{cases} NT \leq (Nrel + Nh_i \times NF_i + 8Nh_i + 2NF_i) \\ NJ \leq (3NF_i + NF_i \times Nh_i + 8Nh_i) \times (Nrel - 1) \end{cases}$

The execution time for each join depends on the join algorithm employed and on the number of attributes being joined at each step. $NA = \max_i(Na_i)$ is the maximum number of attributes being joined at each step. The number of comparisons (which determines changing and adding entries to the database state table) is at most $(NJ \times NA)$. The checking of functional dependencies at each step will be considered separately, and depends on the number of rows being checked, which for any step is at most NT-(Nrel-1). For conditional updates or forced deletions, this check takes approximately the same amount of time (since the number of rows to be checked is the same), denoted Chk1; checking rows that result from forced insertions or forced changes again takes approximately the same time, and will be denoted Chk2.

Assume there are CI conditional insertions, FI forced insertions, CD conditional deletions, FD forced deletions, CC conditional changes and FC forced changes, where the total number of underlying updates is Nrel. This generates an underlying modification table whose size is proportional to $NA \times UMT$, where

$$UMT \leq CI + CD + 2(CC + FD) + 3 \sum_{i=1}^{FI+FC} (Nh_i + NF_i + NF_i \times Nh_i) + \sum_{i=1}^{FC} Nh_i .$$

All entries in this table must be checked when building each tableau T_1, whenever insertions or changes are processed. The upper bounds for checking the side effects of all underlying updates of each type are given by adding up the number of comparisons it takes to compute

$$\Delta_i + \text{number of joins} + \text{check of functional dependencies.}$$

. conditional insertions

$$\text{ECI} \leq \sum_{i=1}^{CI} \{Na_i (Nh^2_i - Nh_i + NF_i) + NA \times (Nrel - 1) + \text{Chk1}\}$$

. forced insertions

$$\text{EFI} \leq \sum_{i=1}^{FI} \{Na_i [Nh_i^2 \times (NF_i + 3) - Nh_i + NF_i] + $$
$$+ NA \times (3NF_i + 5Nh_i + Nh_i \times NF_i) \times (Nrel - 1) + \text{Chk2}\}$$

. conditional deletions

$$\text{ECD} \leq \sum_{i=1}^{CD} \{Na_i \times Nh_i + NA \times (Nrel - 1) + \text{Chk1}\}$$

. forced deletions

$$\text{EFD} \leq \sum_{i=1}^{FD} \{Na_i \times Nh_i + 2NA \times (Nrel - 1) + \text{Chk1}\}$$

. conditional changes

$$\text{ECC} \leq \sum_{i=1}^{CC} \{Na_i [Nh_i (Nh_i + 2) + NF_i] + 2NA \times (Nrel - 1) + \text{Chk1}\}$$

. forced changes

$$\text{EFC} \leq \sum_{i=1}^{FC} \{Na_i [Nh_i^2 \times (NF_i + 5) + Nh_i + NF_i] + $$
$$+ NA \times (3NF_i + 8Nh_i + NF_i \times Nh_i) \times (Nrel - 1) + \text{Chk2}\}$$

For computing Chk1 and Chk2, recall that it is not the whole tableau that is checked for functional dependencies at each step, only the rows involved in a join. A functional dependency is only allowed to modify attributes once, and tableau rows are processed independently. If a dependency is activated after a join (i.e., changes a placeholder variable to parametric), all rows that result from this join will be checked individually, and this dependency cannot cause any subsequent changes. (If for certain rows a given attribute becomes parametric after a join, no subsequent joins can affect this attribute again.)

Assume $NF=\max_i(NF_i)$ dependencies are activated at each of NJ joins. For conditional updates and for forced deletions, the functional dependency check processes at most two rows at each step ($\{t\}_i \cup \overline{w}_{1d}$), and therefore Chk1 takes approximately $(NF \times NA)$ comparisons per join, and thus $O(Nrel \times NF \times NA)$ comparisons per execution. For forced changes and insertions, the functional dependency check processes at most (NT-($Nrel-1$)) rows per join, i.e., $O(NF \times NH)$. Thus, there are $O(NF^2 \times NA \times NH)$ comparisons per join, and hence Chk2 takes $O(Nrel \times NF^2 \times NA \times NH)$ comparisons per execution.

Therefore, since the execution of STEP 2 requires (ECI + EFI + ECD + EFD + ECC + ECI) steps, the algorithm is polynomial in the size of the input. ■

Corollary: Let $NA=\max_i(Na_i)$, $NF=\max_i(NF_i)$, $NH=\max_i(Nh_i)$, and let N=max (NA,NF,NH). If the input size is given by M=$O(Nrel \times N^2)$, the update validation algorithm takes at most $O(M^2)$ comparisons.

Proof

From the expressions given in Theorem II, the worst case occurs when all underlying updates are forced changes, and all relations are subject to both functional dependencies and a template dependency. The input size is $Nrel \times NA(NH+NF+1)$, i.e., $O(Nrel \times N^2)$. Using the results of Theorem II, and abandoning lower-order terms, the following results hold:

STEP1 takes $O(Nrel \times N^3)$

UMT = $O(Nrel \times N^3)$

STEP 2 takes $Nrel \times$EFC (for FC=1), which is approximately equal to $(N^4 \times Nrel + Nrel^2 \times N^3 + Nrel^2 \times N^4)$, and thus it is $O(M^2)$.

Therefore, the algorithm takes at most $O(M^2)$ comparisons, for an input of size M. ■.

Chapter 5

Extending and unifying related work

Several authors have proposed means of analyzing the validity of update translations. This chapter exemplifies how the update validation algorithm proposed in Chapter 3 can be used for obtaining the same results, and how this analysis can be extended to encompass all SJP views.

5.1. Examining some policies proposed by other authors

An example of a proposal that can be described in terms of the update validation algorithm model is that of Keller [KEL82]. According to his model, views involving joins cannot be updated if the attributes that participate in the join do not appear in the view.

Example: [KEL82]

Let R=$\{R_1$<ED,{E→D}>, R_2<DM,{D→M}>}, and $q_v=\Pi_{EM} R_1 \times R_2$. Updating through this view is forbidden, since the join attribute D is not present.

(This example will appear again in the next section, where it will be discussed at length.) This restriction approximately translates to forbidding updates in case of lossy joins or when joins are performed over placeholder variables (i.e., whose values cannot be specified by the user at execution time).

His policy is that of always translating the specified update exactly, even if this causes more changes than desired in the view, being an attempt at liberalizing the restrictions imposed by previous models. The level of view modification achieved is measured in terms of striving for a minimal set of database changes. One of the problems pointed out by Keller is that his minimality criteria involve dynamic decisions as to which underlying updates to perform. This approach resembles forcing updates (since additional operations are allowed) but attributes affected by functional dependencies cannot be forcibly modified.

Recently, Keller [KEL85] liberalized this to the effect of allowing insertions even when functional dependencies are violated. The treatment given is the same as the one proposed in this thesis: changing any existing tuples to agree with the inserted values. He assumes that if a tuple with conflicting values for the functional dependency already exists in the database, it does not appear in the view (otherwise the user would not request insertion of conflicting attribute values). Thus, the effect on the view of inserting a tuple is either to insert the tuple itself (when no violation occurs) or to insert the tuple by changing tuples which did not appear in the view, but already existed in the database and violated the dependency as defined by the insertion. Insertions cannot cause deletions from the view.

In this latter proposal, all underlying relations are in BCNF, the constraints consist exclusively of functional dependencies, all keys appear in the view and joins are performed only over key attributes, so that there exists a unique key for each view tuple corresponding to the key of an underlying BCNF relation. If the update validation algorithm from Chapter 3 processes any view which is analyzed by his model, the result obtained is the same. The existence of view tuple keys is also one of the conditions imposed by Ling [LIN78] for a view to be unconditionally update-viable.

Carlson and Arora [CAR79] are among the few that try extending the constraint domain to multivalued dependencies, to show how this adds complexity to the view update problem. This is done by means of a series of examples, using a scheme over (Employee Dept Project Acct#), containing a subscheme of the form R={R_1<ED,{E→D}> R_2<DPA,{DP→A}>}, and $q_v = R_1 \times R_2$. Part of the instances provided are:

E	D
Jones	D1
Davis	D2
:	:
King	D1

D	P	A
D1	P1	A1
D1	P4	A2
:	:	:
D2	P2	A1

The authors make several comments, among which the following statements are transcribed into symbolic tuple expressions:

a) the underlying conditional policy {(*In*,ED,R_1), (*In*,DPA,R_2)} inserts tuples described by (EDPA) and (eDPA) in the view (the latter for all tuples (eD)

already in R_1), where (EDPA) = (Mays, D1, P3, A5);
b) it is not possible to delete a tuple (EDPA) from the view without deleting all other tuples of the form (eDPA).

Let the placeholder expressions for R_1 and R_2 be respectively (e_1d_1) and $(d_2p_1a_1)$. If the update validation algorithm algorithm is executed for the insertion proposed in (a), the output is
(In,ED,R_1): $\mathrm{ED}p_1a_1$ (is inserted if) $\exists\ p_1,a_1 \mid (\mathrm{D}p_1a_1) \in r_2$ (and $\nexists\ \mathrm{D}' \neq \mathrm{D} \mid \mathrm{ED}' \in r_1$);
(In,DPA,R_2): $e_1\mathrm{DPA}$ (is inserted if) $\exists\ e_1 \mid (e_1\mathrm{D}) \in r_1$ (and $\nexists\ \mathrm{A}' \neq \mathrm{A} \mid \mathrm{DPA}' \in r_2$);
This shows the authors ignored the side effects caused by isolated insertion in R_1: their example on page 417 should also consider tuples (Mays,D1,P1,A1) and (Mays,D1,P4,A2) as a result of inserting (Mays,D1) in r_1 and (D1,P3,A5) in r_2. Since these examples in the paper manipulate relations of 10 or more tuples, it is possible that the reason for this oversight is the size of the relations employed, which makes it cumbersome to derive all side effects. The update validation algorithm, however, reduces this problem to one of systematic manipulation of symbolic tuples, with the consequent conciseness and completeness of results. Arora and Carlson seem to have concentrated in maintaining the view consistency (by checking the derived inter-relation constraint $\mathrm{E} \twoheadrightarrow \mathrm{D} \mid \mathrm{PA}$), without considering that each underlying relation must also be processed.

Their second statement (b) is not correct, either. There is an underlying transformation - (De,ED,R_1) - which achieves deletion of $(\mathrm{ED}p_1a_1)$ (for all tuples $(\mathrm{D}p_1a_1)$ in r_2). This deletes, among others, the desired tuple (EDPA), and yet does not affect the tuples (eDPA) predicted by the authors. In other words, deletion of T=(Mays,D1,P3,A5) can be achieved by deleting (Mays,D1). This does not affect view tuples (Jones,D1,P3,A5) and (King,P3,D1,A5), which they claim *must* be deleted in order to delete T from the view.

5.2. Detecting clean sources

Dayal and Bernstein [DAY82] develop view-trace and view-dependency graphs as a means of detecting, in linear time in the input size (number of attributes and functional dependencies), the presence of *clean sources* for a view update operation, whenever the underlying relations are subject to keys and functional dependencies. (An underlying relation r_i contains a clean source for a view if an update in r_i affects only the desired view tuple(s).) In their graph approach, r_i contains a clean source when there is a path from the node representing r_i to a node representing the view.

Example: [DAY82]
Let R be a scheme over (Employee, Dept, Manager), where $\mathrm{R}=\{R_1<\mathrm{ED},\{\mathrm{E}\rightarrow\mathrm{D}\}>, R_2<\mathrm{DM},\{\mathrm{D}\rightarrow\mathrm{M}\}>\}$, and $q_v=R_1 \times R_2$. If view graphs are built for this specification, there will be a path from relation node [ED] to view node [EDM], and none from [EM] to [EDM], (i.e., R_1 contains a clean source for the view **EDM**, but R_2 does not).

The same fact can also be verified by executing the update validation algorithm. Let (e_1d_1) and (d_2m_1) be the placeholder expressions that represent schemes (ED) and (DM). The operation $(In,\mathrm{ED},R_1,\mathrm{cond})$ will exactly produce a single tuple (EDM_1) (conditional to absence of (ED$'$) in r_1 and existence of a tuple (DM_1) in r_2), whereas (*In*,DM,R_2,cond) creates tuples (e_1DM) in the view (if (e_1D) exists in r_1). Similarly, (*De*,ED,R_1,cond) causes deletion of (EDM_1) from the view, whereas (*De*,DM,R_2,cond) deletes the set (e_1DM) from the view. Therefore, relation R_1 contains a clean source for insertion (and deletion) of view tuples (EDM). Similar remarks, for the same example, are made by Keller [KEL82] (see previous section).

Like Dayal and Bernstein's graphic method, the second step of the update validation algorithm also runs in linear time (in number of relations and functional dependencies per relation) for all cases considered by the view graphs which do not involve inter-relational constraints: since only functional dependencies are involved, and forced updates are ignored, each relation is represented by a single row in the initial tableau T_1, and the number of joins is at most $Nrel-1$.

Dayal and Bernstein also propose update programs (which approximate complete policies u_v), and define conditions under which these programs exactly perform the desired update. For insertions, for instance, these conditions are that the primary keys of all relations must appear in the view, these keys cannot have NULL values, and insertions must not violate the functional dependencies. If this statement is translated into the symbolic input for the update validation algorithm, for view **EDM**, it corresponds to the policy $<$*In*,EDM,{(*In*,R_1,ED), (*In*,R_2,DM)}$>$. The symbolic output indicates that an exact translation occurs only if tuples of the form (e_1D) do not exist in r_1, or if (DM) already exists in r_2. This means, in fact, that if (DM) does not yet exist in r_2, and r_1 contains tuples (e_1D), insertion of (EDM) in the view will also be accompanied by insertion of (e_1DM) in the view. In other words, this may not be an exact translation.

On the other hand, for a view function $q_v = \Pi_{EM} R_1 \times R_2$ (see the same example from Keller in the previous section) there is no exact translation provided by the authors, since the attribute **D**, which is a key for R_2, does not appear in the view. Keller [KEL82, KEL85] does not allow this update either, because **D**, a join attribute, does not appear in the view. Such is not the case for the update validation algorithm, as it allows parametric specification of variables that do not appear in the view. The policy $<$*In*,EM,{(*In*,ED,R_1), (*In*,DM,R_2)}$>$ can again be used in this case. The value of parametric variable 'D' is defined at run time by the system (e.g., taken from the user's access code). Thus, an exact translation can be provided even if the key of a relation (or a join attribute) does not appear in the view.

As also pointed out in [DAY82], it is not always possible to find a clean source to update a view. Absence of clean sources translates, in the view graph model, to absence of paths from any underlying relation node to the view node. View graphs, however, do not describe which side effects may occur in the absence of clean sources, whereas execution of the update validation algorithm provides this additional information. This allows the designer to choose the more appropriate translation (e.g., the one where "minimal change" occurs in the database, as desired by [DAY82]).

Given the database scheme R=$\{R_i <\{A_{i_j}\},\{C_{i_n}\}>\}$, the detection of clean sources for an update operation OP involving view tuple Γ_v can be achieved by the following algorithm based on the update validation algorithm:

Algorithm I — Detection of clean sources

1) Execute the update validation algorithm with input $<R,q_v,u_v>$, where

$$u_v=<\mathrm{OP},\Gamma_v,\{\mathrm{OP},\{t\}_i,R_i,\mathrm{cond}\}>.$$

The parametric variables of $\{t\}_i$ are $(\{A_{i_v}\},\{\tilde{A}_{i_v}\})$, where A_{i_v} are all attributes of R_i that appear as parametric variables in Γ_v; $\tilde{A}_{i_v}$ correspond to all attributes that do not appear in the view and yet are assigned specific values by the system. All underlying updates must be of the same type as the desired view operation.

2) Analyze the output of this execution:

- if the result of isolated updating of some relation r_i exactly represents the desired update $\mathrm{OP}(\Gamma_v)$ (i.e., the symbolic output expression is equivalent to Γ_v), then r_i contains a clean source for the operation;
- if no such result occurs, there is no clean source for the operation.

The reasoning behind Algorithm I is that, in order to affect exactly the indicated view tuple T_v, the only underlying tuples to be considered are those whose attributes match specific (parametric) values of the view attributes. The next example shows that detection of clean sources is thus made possible for a larger class of constraints.

Example: Let $R=\{R_1<AB,\{A\rightarrow B\}>$, $R_2<ACD,\{*[AC,AD,CD]\})$, and $q_v=\Pi_{ABD}R_1\times R_2$. The placeholder expression for R_1 and the template for R_2 are

A	B		A	C	D
a_1	b_1		a_2	c_1	d_1
			a_2	c_2	d_2
			a_3	c_1	d_2
			a_2	c_1	d_2

R_2 contains a clean source for deletion of one tuple from the view, since deletion of Γ_v=(ABD) is exactly accomplished by deletion of (AcD) from r_2.

5.3. Forcing updates

Another advantage provided by the update validation algorithm is that it takes into consideration actions to be taken when exceptions occur. Chapter 3 shows that forcing changes or forcing insertions in relations subject to functional dependencies may eliminate tuples from the view. Fagin, Vardi and Ullman [FAG83], who suggested forcing updates, exemplify this type of action by updating the scheme (Employee Child Dept) of the form $R=\{R_1,<ECD,\{E\rightarrow D\}>\}$, $q_v=\Pi_{ED}R_1$.

Let the relation and view extensions be

E	C	D		E	D
Gauss	Yoni	Math		Gauss	Math
Turing	Yoram	CS		Turing	CS
Turing	Gabi	CS			

If there is forced insertion of (Turing,Math) (which violates the dependency E→D), the view is changed to {(Gauss,Math), (Turing,Math)}.

This operation corresponds to the rule $<In$, ED,$\{In$,ECD,R_1,force$\}>$, whose initial tableau T_1 is

E	**C**	**D**	**[E**	**C**	**D]**
(E	c_1	D,	[E	c_1	D′])
E	C	D			

The first entry in this tableau indicates that the set of tuples [Ec_1D′] may need to be eliminated and replaced by (Ec_1D), because of the functional dependency E→D. The output (ED,[ED′]) agrees with the authors' result: (ED) is inserted, but (ED′) may disappear from the view.

5.4. Analyzing effects on other views

Until now, the issue of view interference (i.e., when updates to a view modify the extension of other views) has not been considered in this thesis. The update validation algorithm can be extended to determine the effects of the underlying operations on any other view V′, by applying the corresponding generating function $q_{v'}$ to the each tableau T_1. Thus, after T_1 is created, the effects of any update rule on other views can be observed:

Algorithm II — Determining side effects

1) Execute the update validation algorithm with input $<$R,$q_{v'}$, $u_v>$
2) The output describes the effects on V′ of the update policy u_v.

Update propagation can be measured by examining the complementary view. Bancilhon and Spyratos [BAN81] postulate that desirable updates are those that should leave the complement invariant. In [FAG83], using the same example of the previous section, it is also pointed out that the complementary view $\bar{V}$ given by $q_{\bar{v}}=\Pi_{EC}R_1$ is not affected. Even though true for this particular example, this may not always be the case: the complement's state before the update is $\bar{V}=\{(e_1c_1)\}$ (which includes tuples corresponding to the expression (Ec_1)) (if $\exists$ c_1,d_1 | $(Ec_1d_1) \in r_1$); after the update, the result is $\bar{V}=\{$(Ec_1)$\}$ (if $\exists$ c_1,d_1 | $(Ec_1d_1) \in r_1$ - which will be true if the update is accomplished). In the following tableau, which is similar to a tableau query, the first ("summary") rows represent the effect on the complementary view $\bar{V}$ resulting from the update.

E	**C**	**D**	**[E**	**C**	**D]**
E	c_1		[E	c_1]	
E	C				
(E	c_1	D	[E	c_1	D′])
E	C	D			

The tuples in the complementary view will not be affected only if some (Ec_1D′) already existed and $c_1=C$. In particular, if the relation is initially empty and (ECD) is such that C happens to be a null value, the complementary view may not be affected (if tuples with nulls are not considered in the view), even though the complementary *projection* will contain a new tuple. This example has been frequently used in the literature as an instance of an invariant complement. Due to the liberalization of policies advocated here, an update through view **ED** can actually manipulate the contents of attributes not in the view (i.e., C), by specifying an underlying translation that assigns a parametric variable to such attributes. If this translation is accepted, **(EC)** ceases to be an invariant complement.

Detecting whether a complement remains invariant under an update policy u_v is accomplished by the following algorithm, which is an extension of Algorithm II for a special class of views - that of complementary views:

Algorithm III - Invariance of complements

1) Determine the symbolic format of the complementary view tuples $\{t_{\bar{V}}\}$ before the update;

2) Execute the update validation algorithm with input $<R, q_{\bar{v}}, u_v>$, where $q_{\bar{v}}$ is the complementary view generating function.

3) The complementary view is invariant for each of the following cases:

a) the output expressions {E} are equivalent to $\{t_{\bar{V}}\}$, or

b) the output is of the form (w,[w']), w is equivalent to w', and nulls are ignored in the view; or

c) there are no user-entered parametric variables in the output, neither are such parametric variables eliminated by projection (e.g., $q_{\bar{v}} = \Pi_{[R_1 \cup R_3]} R_1 \times R_2 \times R_3$ may result in a placeholder-only row if the update is performed in R_2, and yet the complement may be affected); or

d) the output rows of $\bar{V}$ are determined by selection operators which refer to sets of attributes disjoint from those forming the view V. In this case, either the view V is updated, or the complement is updated.

If none of the above cases occurs, the invariance of the complement will depend on the underlying state of the database.

Example: Let $q_v = \sigma_{B>10} R_1 \times R_2$, defined for the scheme $R = \{R_1 <AB, \{A \rightarrow B\}>, R_2 <BC, \{B \rightarrow C\}>\}$. If there are no dangling tuples, a possible complement is $q_{\bar{v}} = \sigma_{B \leq 10} R_1 \times R_2$. Let $<In, ABC, \{(In, AB, R_1, cond)\}>$ be a policy proposed for view V. The result of this policy is

(ABC_1) is inserted in the view V if
$[\exists\ B \mid (BC_1) \in r_2, B \rightarrow C_1 \wedge B > 10] \wedge \nexists\ [B' \neq B \mid (AB') \in r_1]$.

For the complementary view, the same result is given, except that $B \leq 10$. Therefore, $\bar{V}$ is invariant under this operation.

The policy $\leq In, ABC, \{Ch, (AB, AB'), R_1, force\}>$ will produce the following results for V and $\bar{V}$:

in V: (ABC_1) is inserted if $\exists\ B \mid \{(BC_1) \in r_2, B \rightarrow C_1 \wedge B > 10\}$ and
$(AB'C_2)$ is deleted if $\exists\ B' \mid \{(B'C_2) \in r_2, B' \rightarrow C_2 \wedge B' > 10\}$.
in $\bar{V}$: the same result, but for values of B and B' less than or equal to 10.

For this second policy, nothing can be guaranteed, because even if a user-entered value $B > 10$ ensures no insertion occurs in $\bar{V}$, the initial state for r_1 may contain (AB') for $B' \leq 10$. In this case, insertion of (ABC_1) in V causes deletion of $(AB'C_2)$ in $\bar{V}$.

5.5. The updatable view - supporting multiple translations

Keller [KEL82] avoids some translation semantics because "they can have arbitrary complexity". By this he means not only that a unique translation is not usually possible, and the user's intention may not be correctly interpreted, but also that correctly expressing the desired interpretation presents too many problems. To support this claim, he gives the following example: For $R=\{R_1<ED,\{E \rightarrow D\}>, R_2<DM,\{D \rightarrow M\}>\}$, and $q_v=R_1 \times R_2$, the request to "delete from EDM where D=Toys " can be interpreted as either a deletion from relation ED of the tuples with D='Toys', or as a deletion from relation ED of the tuples where D='Toys' followed by a deletion from relation DM of the tuple where D='Toys'.

The first objection, non-uniqueness of translation, can be dealt with by observing that if the translation is defined at design time, there is no possibility of doing "more" than intended, since by execution of the update validation algorithm the result of the underlying updates is always clear. Secondly, the semantics of each request can be translated into a different conditional policy:
DEL1: $<De$,eDM, $\{(De$,eD,$R_1)\}>$
DEL2: $<De$,eDM, $\{(De$,eD,$R_1)$, $(De$,DM,$R_2)\}>$.

Both policies delete all tuples of the form (eDM). However, after DEL1 is performed, insertion of a tuple (ED) into r_1 might insert tuple (EDM) in the view; after DEL2, this insertion would not appear in the view. Yet another interpretation (not mentioned by Keller) is possible:
DEL3: $<De$,eDM,$\{(De$,DM,$R_2)\}>$, which only deletes from the (DM) relation. This example shows that the symbolic tuple notation can help express many policies whose interpretation is sometimes not clear.

If the burden of translating updates is transferred to each view, then Spyratos' suggestion of an operational view model [SPY82 - see Chapter 1] can be extended. Spyratos suggests that views be defined according to the way updates affect data and the allowed updates do not interfere with his "definition sets" partition. The updates allowed by the operational model proposed here are all those that are part of the view definition. This corresponds to the abstract data type approach of Tucherman, Casanova and Furtado [TUC83 - see Chapter 1]. Therefore, each view forms its own definition set, and checking whether an update is allowed is transformed into checking whether the corresponding policy "procedure" has been authorized for the particular view. In this interpretation, views should be treated as single-relation updatable units, defined by means of two different sets of operations: view creation and view update mechanisms. The former describe the sequence of operations involved in creating the query image, and correspond to the standard concept of view definition; update policies (u_v) describe all valid elementary operations on the particular view, and their translation. Since every view contains the definition of valid updates through it, validating updates at execution time becomes checking whether the operation is acceptable for the given view, which reduces the execution overhead of checking for undesirable effects on other views. Traditionally, each update operation is translated in a unique way; in the framework suggested, views which in the standard approach would be considered equivalent (i.e., returning the same image to a query) may have different interpretations for update requests, becoming, therefore, different views, increasing the operational flexibility.

A view is thus defined by the triple

$$(\mathrm{R}, q_v, U_v),$$

where $U_v = \{u_v\}$ is the set of permissible updates for the given view. Valid updates, furthermore, are those that can be expressed by composition of elements of U_v. If this concept is implemented, the view designer is allowed more control over the degree of changes to the view and to the database. Multiple update translations coexist in a view by allowing each interpretation to have a different procedural name. It is conceivable that, for specific applications, the user be allowed to choose at run time the correct interpretation desired.

Example: Let R=$\{R_1$ (ECD,{E→D>}, $q_v = \Pi_{ED} R_1$ (the example of section 5.2), and let the following policies exist in the view:

INS1: $<In$,ED,(In,ECD,R_1,force}>
INS2: $<In$,ED,(In,E"NULL"D,R_1,cond}>
DEL1: $<De$,eD,(De,ecD,R_1,cond}>
DEL2: $<De$,ED,(De,EcD,R_1,force}>

Let the relation and view extensions be, respectively

E	C	D
Gauss	Yoni	Math
Turing	Yoram	CS
Turing	Gabi	CS

E	D
Gauss	Math
Turing	CS

The following examples show the effect of the operations, on this initial state:

INS1(Turing,Math):

E	C	D
Gauss	Yoni	Math
Turing	Yoram	Math
Turing	Gabi	Math
Turing	C	Math

E	D
Gauss	Math
Turing	Math

where the value of 'C' in the last tuple of (ECD) is determined by the system.

INS2(Turing,Math): not accepted, since it violates the dependency E→D;

INS2(Einstein,Phys):

E	C	D
Gauss	Yoni	Math
Turing	Yoram	CS
Turing	Gabi	CS
Einstein	NULL	Phys

E	D
Gauss	Math
Turing	CS
Einstein	Phys

DEL1(Math):

E	C	D
Turing	Yoram	CS
Turing	Gabi	CS

E	D
Turing	CS

DEL2(Turing,CS):

E	C	D
Gauss	Yoni	Math

E	D
Gauss	Math

Yet another advantage of this operational approach is that it can be analyzed under the complement model, which is also the idea behind definition sets. One problem intrinsic to the complement mapping approach is that invariant complements are hard to find, and the constraints imposed on them virtually preclude the possibility of executing any view updates. By applying the updatable view model, the rigidity of the rules surrounding invariant complements disappears: instead of describing the updatable view as the one with a unique complement to remain invariant at all times, the designer can define different sets of updates based on which complement will remain invariant (always keeping in mind that a view can have many different complements).

Thus, instead of specifying a single (and possibly even unupdatable) view, the designer can define the same view formation function several times, for each different set of allowable update policies. Even though the (unique) invariant complement is lost, for practical purposes the complement of each view is invariant with respect to the set of updates allowed on that view. The query image for the set of views may be the same at all times, but the views are different: under the abstract data type approach, the views differ because the set of operations allowed is different; under the complement mapping approach, the views differ because different complements remain invariant for each of the operations defined.

As remarked in Chapter 1, Casanova and Furtado divide approaches to view updates as either generalized mappings or abstract data type. If seen under this new light, however, the division ceases to be so marked.

This chapter showed some examples of how the update validation algorithm proposed in Chapter 3 can be used to interpret and expand results obtained by other update validation tools. Some suggestions for further extensions are discussed in the next chapter.

Chapter 6

Summary and directions for future work

6.1. Summary

This thesis presents a new approach to the view update problem, that allows the designer to test the outcome of a large class of view update policies on SJP views. It gives the designer the opportunity of defining many different translations for the same update operations, and actions to be taken when exceptions occur. Previous research on database updates has concentrated on unique, minimal and unambiguous translations on updates through views. Here, a new framework was presented - that of liberalizing mapping restrictions. Not only does this approach enhance the flexibility in updating views, but it also enlarges the set of views which can be updated. Under this premise, an algorithm to validate arbitrary update policies was presented, which predicts all possible side effects that can occur as a result of a given update mapping.

The proofs of Chapter 4 show that the algorithm is correct for any SJP views. Except for the proof of correctness of the $Ex(q_v)$ rules (lemma 7.1), all other proofs rely only on the monotonicity of q_v. The algorithm is, therefore, applicable to views formed by monotonic functions other than select, project and join operators. This just requires extending the execution rules in $Ex(q_v)$.

Furthermore, this algorithm makes the designer aware of the role played by individual underlying relations, which often seems to be disregarded in analyses of update effects. The use of symbolic tuple expressions shows how implementation of update requests can be simplified so as to indicate only the attributes that need to be affected. Not only does this standardize the user's requests, but also saves execution time, since there is no need to check characteristics of attributes which are used only in the internal mapping (corresponding to the placeholder variables).

The main contributions of the thesis belong to three categories: the presentation of a liberalized approach to database updates; the development of a systematic and error-free algorithm to predict update effects, which generalizes and unifies previous attempts in that direction; and the characterization of types of constraints that are conducive to fostering undesirable side effects.

The analysis presented here of the update problem under the "liberalization" assumption is diametrally opposed to the formal analyses previously published. As an immediate consequence, more update mappings can be shown to be acceptable, and more views can be successfully updated, including views not supported by universal relation databases, and even lossy views, which are avoided by all authors. However, as mentioned by Biskup and Bruggeman [BIS83], there is no reason why lossy views should not be considered, as long as the user is aware of the possible implications in using them. The enhanced set of updatable views includes many which, in general, have been considered to be query-only views [FUR79, ARO80, DAY82, KEL82, SPY82, MAS84, KEL85] (e.g., allowing reference to sets of attributes not visible in the external schema). Other marked differences between previously allowed mappings and the policies analyzed here include comprehensive treatment of tuple replacement operations as a distinct type of update request, differentiation and combination of forced and conditional updates, and combination of different types of underlying operations to translate a given external request.

The update validation algorithm can be seen as a method of syntactically determining the conditions under which the semantic consistency of a database will be maintained. It allows systematic and error-free prediction of all possible side effects a policy may have, associated with the underlying state where such side effects occur. This prediction is extended to any SJP views in the database so that the effects of a policy on another view can also be determined. Chapter 5 shows how the update validation algorithm generalizes other view design tools, and provides a unifying framework for analysis of approaches to the update problem which have been until now considered independent and incompatible.

Finally, whereas most analyses of update validation restrict the set of integrity constraints to functional dependencies when keys are present in the view [LIN78, ARO80, DAY82, KEL82, BRO85, KEL85], the update validation algorithm uses join dependencies, extending thereby the constraint domain. Previous analyses of properties of tableaux, recursive axioms and template dependencies (e.g., [AHO79, MIN83, SAG85]) were concerned with query derivation, while here these properties were interpreted under the framework of characterization of side effect propagation. Chapter 3 showed that simple template dependencies can interact with functional dependencies so as to generate conflicting information when insertions are forced, and necessary and sufficient conditions were given for which this does not occur. Furthermore, non-simple template dependencies were shown to be generators of infinite chains when forced updates are requested. Thus, the designer can be made aware of some types of constraints which may foster either side effect blow-up or ambiguous updates, *independent* of the type of update requested.

Suggestions for future work fit into three main categories: extensions to the update validation algorithm, suggestions for its use as a design tool, and some open topics in the view update problem.

6.2. Extending the update validation algorithm to automate the check for the desired update outcome

The output of the update validation algorithm describes all possible results for the policy being tested, so that the user can verify whether the intended update is, in fact, reflected in the view. It is possible, however, that this checking may be made cumbersome by a multiplicity of side effects, allied to a number of complex underlying conditions.

This check can be automated by adding a third step to the algorithm, which indicates as its output whether the specified operation is actually implemented.

```
STEP 3
 DETERMINE EFFECT OF THE ACTUAL POLICY PROPOSED
 Let the policy be $<OP,\Gamma_v,\{u_v\}>$
 If the desired update $\Gamma_v$ appeared among the result rows in STEP 2, stop.
 Else do using the set $\{u_i = (Op_i,\{t\}_i,R_i,exception)\}$
 1. Eliminate from the underlying modification table information which is not pertinent to the specific update requested $\Gamma_v$
 Let $\theta_i$ represent the set of expressions in $R_i$ that may contribute to forming $\Gamma_v$
 1.1 if OP = 'insert' do for each underlying update $u_i = (Op_i,\{t\}_i,R_i,exception)$
     1.1.1 If $Op_i = Ch$ or $In$, eliminate all deletions $[\overline{w'}_j]$. Add to the database state description associated with $\theta_i$ the clause $\theta_i \neq \overline{w'}_j$ (i.e., no insertion can consider this set of tuples).
     1.1.2. If $Op_i = De$, the set of rows to be used is the one which remains unaffected by the deletion (i.e., $\theta_i \neq (\{t\}_i \cup \overline{w}_{1d})$ is added to the database state description).
 1.2 if OP = 'delete' do for each underlying update $u_i = (Op_i,\{t\}_i,R_i,exception)$
     1.2.1 If $Op_i = Ch$ or $In$, eliminate all insertions $\{\overline{w}_j\}$ and associated database state description. If this results in $In(\{t\}_i,R_i,force) : \Delta_i = \emptyset$, use a placeholder expression for $R_i$.
 2. Define the tableau $T_1$ as being this reduced underlying modification table, thereby processing all underlying updates simultaneously
```

3. Perform the remainder of this step as in STEP 2, stopping as soon as any output row translates to the desired update Γ_v. For changes, each join of rows of the form (new tuple, [old tuple]) results in a new row where the first (respectively, second) element is obtained from joining the new (old) elements of the original rows. Joining two different user-entered parametric variables is forbidden.

The cost of executing this third step is bound by the number of row joins. For conditional update executions, and deletions over relations subject exclusively to functional dependencies, the number of row joins is $Nrel-1$, since each relation is replaced by a single row.

For all other cases, however, the number of joins may grow exponentially, since each relation can be potentially replaced by a set of rows, and all sets have to be joined together. The exponential number of join operations is a characteristic inherent to the problem of generating queries from underlying relations. Since this step is only used to check if the desired update occurs, most rows can be abandoned after each execution of $Ex(q_v)$ if subsequent operations on these rows cannot create the intended result row. This can be verified by checking the row's attributes against the corresponding attributes in the desired expression Γ_v. In practical applications, appearance of too many result rows in STEP 2 corresponds to a multiplicity of possible side effects, which may suggest that the policy proposed should not be implemented.

6.3. Improving the execution time of the update validation algorithm

Some improvement may be achieved by eliminating unnecessary rows during tableau processing. A slight improvement is achieved by noting that relations that do not contribute to view formation need not be considered. This, however, only decreases the execution time by reducing Nrel, and saves storage space in eliminating some tableau rows.

Simplifications can also occur when special dependencies are known to apply. For instance, when $\{C_{i_n}\}$ contains a Y-partial template dependency, and forced changes are applied to attributes in R_i - Y, no insertions (of $\{t\}_i$ or $\overline{w}_{1c}$) need to be pushed down the template (and thus $\{(\overline{w}_{s+1})_j{}^B\} = \{(\overline{w}_{s+1})_j{}^G\} = \emptyset$). The number of comparisons in $Ch(\{t,t'\}_i, R_i, force)$ is then reduced from $O(Na_i \times Nh_i{}^2 \times NF_i)$ to $O(Na_i \times NF_i)$.

Another special case occurs when the underlying set of operations consists exclusively of deletions, or of insertions that do not force functional dependencies (i.e., there is no mix of insertions and deletions in the policy). In this case, the underlying modification table need not be stored, and each set in $Op(\{t\}_i, R_i, exception)$ is generated when each tableau T_1 is built (since the underlying modification table is used only to determine when deletions prevent insertions). This saves approximately $O(Nrel \times N^3)$ space.

Finally, if q_v is an optimized query function, then the number of joins may also be minimized. This is not always true, since minimization often involves executing select operators as soon as possible (which decreases the number of tuples that participate in later joins). For the update validation algorithm, this makes no difference, since assigning a selection parametric value to an attribute does not decrease the number of symbolic expressions that may participate in joins.

6.4. Extending the analysis to other normal forms

Whereas most authors restrict themselves to analysis of BCNF relations, this thesis relaxed this assumption to BCNF partitions. A suggested extension would be therefore the generalization to other types of functional dependency interaction. This section presents some partial results that limit the level of liberalization one can expect in this direction.

Proposition:

Consider a relation scheme containing attributes (XYA), with dependencies X→A and Y→A, such that X↛Y and Y↛X. Processing a request for forced insertion (change) for this scheme generates an infinite number of pairs $(w_{fd},[w'_{fd}])$.
Proof

Insertion of a tuple containing (XYA) generates

$$w_1 = (Xy_1A,[Xy_1A']) \text{ and } w_2=(x_1YA,[x_1YA'])$$.

Since Y is a left hand side, insertion of (Xy_1A) may disagree over YA with some other existing tuples, and therefore this insertion requires another change level

$$(x_2y_1A,[x_2y_1A''])$$.

Insertion of (x_2y_1A) may disagree with yet other tuples over XA, thus requiring another pair of changes $(x_2y_3A,[x_2y_3A'''])$, etc.

Notice that w_2 will have to undergo the same process independently. This type of functional dependency interaction generates two chains of changes. Both chains can grow arbitrarily, as long as new symbols are generated for X or Y, and can only be processed finitely if the relation's extension is part of the input. □

By the same type of proof mechanism, it can be shown that each level of functional dependency transitivity (e.g., X→A→B→C→...) requires one additional set of pairs $(w_{fd},[w'_{fd}])$. Furthermore, if part of a left hand side is dependent on some attributes of its right hand side (e.g., XY→A and A→Y), at least three levels of substitution are required. If these two results are combined, several types of functional dependency interaction can be shown to produce unbounded chains. These results impose limits on the set of functional dependencies that can be maintained, given forced insertions or forced changes.

6.5. Extending the analysis to other types of template dependency

This thesis restricted the analysis of template dependencies to simple typed minimal template dependencies, which Sagiv [SAG85] has proved correspond to join dependencies. The restriction of constraints to functional dependencies and join dependencies follows the assumptions of several authors when analyzing models of real world database design. For updates, this presents the added advantage that there is no generation of deletion or insertion chains. Chapter 3 has an example of how maintaining non-simple template dependencies can give origin to insertion chains of unbounded length.

Chapter 4 shows that simple template dependencies require only one level of substitution to derive all possible updates which may be necessary to maintain consistency of a relation. The same type of proof mechanism is used by Minker and Nicolas [MIN83], when showing that query answering in deductive logic databases will not result in infinite derivation paths if the clauses to be substituted correspond to what they define as *singular* recursive axioms. The class of singular recursive axioms properly contains the class of simple typed template dependencies.

Sagiv [SAG85] uses a method similar to applying substitution rules upwards to define the composition of a template dependency T with itself, denoted $T \circ T = T^2$. It can be shown that the rows of T^2 correspond to the expressions that are obtained as additional insertions if two levels of substitutions are required in a chain (i.e., if insertion of $\{(\overline{w}_{s+1})_j\}$ requires further insertions). He proves that $T \equiv T^2$ if and only if T is a minimal join dependency tableau. Thus, only for join dependencies can one always be sure that generation of additional updates takes exactly one level of iterations. This means that, in the general case, the computation of additional updates may never end, unless one is allowed to process the database itself as part of the input, rather than its symbolic scheme description.

This is a problem that has been recognized in the context of query answering in deductive databases. As is pointed out by Gallaire *et al.* [GAL84], inference methods work well when no recursive axioms (e.g. template dependencies) are present, otherwise termination presents a problem. Inference methods for logic database access must use an interleaving of deductive laws and searching for facts in the extensional database.

Thus, another extension to the work presented in this thesis is that of trying to analyze the influence of non-simple template dependencies in update propagation. This would enhance the role of the update validation algorithm as a tool to guide the view designer in choosing sets of integrity constraints less liable to cause unwanted side effects. The designer would thus be able to obtain information on update propagation even before proposing any specific policy. The constraints' characteristics would become an indicator of the type of side effect that might occur.

The two sections that follow analyze some results in this direction. It will be shown that, in case of deletions, the update validation algorithm can be extended to manipulate any template dependency in polynomial time, as long as it maintains the policy of choosing $\overline{w}_{1d}$ for each additional mapping in a deletion chain. For insertions, a subset of the class of general template dependencies (composed of those that contain an embedded simple minimal template dependency) is analyzed. It will be shown which insertion sequences in an insertion chain will eventually stop, without creating the need for additional updates. Both forced insertions and deletions can result in an unbounded set of additional operations. It will be shown that the chain of additional deletions can be described in a compact form. Additional insertions, however, besides constituting a potentially infinite set, are also characterized by the fact that the description of an insertion chain has unbounded growth as well.

6.5.1. Handling deletions for any type of template dependency

Proposition:

There exists a modified version of the update validation algorithm that runs in polynomial time in number of comparisons, independent of the type of template processed, if the set of underlying operations consists exclusively of deletions and any conditional updates.

Proof:

Conditional operations do not require propagation of updates. Insertions can generate an unbounded number of insertion expressions. It will be proved here that, even though deletions can also have the same type of ripple effect, this effect can be described in a recursive procedure, and this procedure can be derived in most two template substitutions - i.e., an additional $O(Nh_i \times Na_i)$ operations per relation scheme. Thus, even though the output of the update validation algorithm cannot enumerate all possible deletion side effects, it can be modified to describe these side effects in a procedural form in polynomial time. Therefore, deletions can be processed in polynomial time, independent of the type of template dependency defined in $\{C_{i_n}\}$. The mapping that checks whether a deletion should be propagated is built by creating at each iteration a new copy of the template dependency (where no symbol has appeared before), and replacing the conclusion row of this copy by the expression describing tuples that must be deleted.

Let $\{t\}_i \cup \overline{w}_{1d}$ represent the set of tuples that must be deleted to achieve deletion of $\{t\}_i$, for *any* template dependency, where $\overline{w}_{1d} = \overline{w}_k$ (i.e., $\overline{w}_{1d}$ corresponds to the substitution of row w_k). The check of whether deletion of $\overline{w}_{1d}$ requires further deletions is done by generating a new template, replacing its conclusion row by $\overline{w}_{1d}$, and pushing $\overline{w}_{1d}$ up. Call the mapping thus generated ϕ_1. If ϕ_1 is such that some row $\phi_1(w_j)$ is subsumed by $\overline{w}_{1d}$, then $\overline{w}_{1d}$ is effectively deleted, and the set $\{t\}_i \cup \overline{w}_{1d}$ is sufficient.

Assume, therefore, that there exists no row $\phi_1(w_j)$ subsumed by $\overline{w}_{1d}$. Let Same(k,s+1)=K1. Since ϕ_1 was generated by replacing w_{s+1} by $\overline{w}_{1d}$,

$$\Pi_{K1}\{t\}_i = \Pi_{K1}\overline{w}_{1d} = \Pi_{K1}\phi_1(w_k).$$

All other symbols in $\phi_1(w_k)$ will be new. Any parametric variable in ϕ_1 must correspond to some parametric variable in $\overline{w}_{1d}$. Since $\overline{w}_{1d}(=\overline{w}_k)$ was the first row found with parametric variables in the first substitution, then $\phi_1(w_k)$ will also be the first row with parametric variables in ϕ_1. Thus, $\phi_1(w_k)$ is the deletion chosen to guarantee deletion of row $\overline{w}_{1d}$. The remaining rows $\phi_1(w_j)$ will match $\overline{w}_{1d}$ in Same(j,k+1), and all other symbols will contain new values.

The same symbol pattern will occur in the next substitution mapping ϕ_2, where $\phi_2(w_k)$ will be chosen to guarantee deletion of $\phi_1(w_k)$, (which will have replaced the conclusion row). Symbols which are "carried over" from ϕ_1 to ϕ_2 are those corresponding to Same(j,s+1), for j=[1..s]. It is easy to see that, if the deletion of $\overline{w}_{1d}$ was not sufficient, neither will deletion of $\phi_1(w_k)$ be sufficient, because both substitutions will be characterized by the same symbol pattern, and correspond to different sets of tuples. Since by the hypothesis no row $\phi_1(w_j)$ is subsumed by $\overline{w}_{1d}$, then no row $\phi_2(w_j)$ is subsumed by $\phi_1(w_k)$.

Consider the template that describes the i-th additional deletion in this process, corresponding to mapping ϕ_i. Generate a new template from the original template dependency by adding to each symbol the superscript i. Replace the conclusion row by the expression $\phi_{i-1}(w_k)$. Apply substitution rules. The resulting template will correspond to the database state description for which deletion of $\phi_i(w_k)$ will be necessary to maintain consistency. The symbols of $\phi_{i-1}(w_k)$ are given by: $\Pi_{K1}\phi_{i-1}(w_k) = \Pi_{K1}\{t\}_i$; the remaining symbols in $\phi_{i-1}(w_k)$ are given by the corresponding symbols in $\phi_i(w_k)$ with one subtracted from the superscript.

Therefore, the ripple effect of a forced deletion can be represented by $\{t\}_i \cup \overline{w}_{1d}$, together with the (recursive) rule:
ϕ_1: $\phi_1(w_k)$ (is deleted) for the database state description given by replacing w_{s+1} by $\overline{w}_{1d}$ and pushing $\overline{w}_{1d}$ up the template;
ϕ_i: $\phi_i(w_k)$ (is deleted) if $\phi_{i-1}(w_k)$ (is deleted) and the database state description given by the rows of mapping ϕ_i describe existing tuples.

Even though this generates an unbounded set, its description can be computed in two template substitutions. The recursive description of the deletion ripple effect is possible only because $De(\{t\}_i, R_i, force)$ chooses always the first row where a parametric variable occurs (whereas each insertion expression can potentially generate Nh_i new insertions). □

Example: Consider forcing the deletion of (ABC) for the template on the left. The first substitution generates the template on the right, and $\overline{w}_{1d} = (a_1 b_1 C)$ $(w_k = w_1)$.

A	B	C
a_1	b_1	c_2
a_1	b_2	c_1
a_2	b_1	c_1
a_2	b_2	c_2

A	B	C
a_1	b_1	C
a_1	B	c_1
A	b_1	c_1
A	B	C

The next set of templates show the mappings ϕ_1 and ϕ_i. Notice that if deletion of $\overline{w}_{1d}$ is not sufficient, neither will deletion of $\phi_{i-1}(w_k)$ be sufficient to maintain consistency.

A	B	C
${a^1}_1$	${b^1}_1$	C
${a^1}_1$	b_1	${c^1}_1$
a_1	${b^1}_1$	${c^1}_1$
a_1	b_1	C

A	B	C
${a^i}_1$	${b^i}_1$	C
${a^i}_1$	${b^{i-1}}_1$	${c^i}_1$
${a^{i-1}}_1$	${b^i}_1$	${c^i}_1$
${a^{i-1}}_1$	${b^{i-1}}_1$	C

Since deletions do not violate functional dependencies, if this procedural approach is taken, the update validation algorithm can analyze the result of underlying updates for any type of functional dependencies and template dependencies.

6.5.2. Analyzing insertion chains for more general classes of template dependencies

This section describes some of the properties of insertion chains for a subset of the class of general typed template dependencies, and characterizes the conditions under which redundant insertions may appear in a chain. The subset analyzed corresponds to all non-simple template dependencies such that, if they contain an embedded simple template dependency, then this embedded template dependency must be minimal.

By the definition of necessarily redundant insertion expressions given in the first section of Chapter 4, insertions for partial dependencies need only be checked for the set of attributes corresponding to the embedded constraint. Thus, unless otherwise specified, all template dependencies in this section will be assumed to be full dependencies. Recall (Chapter 3) that when an insertion expression E replaces a hypothesis row w_k and substitution rules are applied, a mapping σ is generated where $\sigma(w_{s+1})$ is a new insertion expression. This is denoted by σ: $w_k \leftarrow E$.

Let E be a new insertion expression generated in a chain. If pushing E down a template results in E', E' is *not* a new insertion expression if it can be generated without using E. This section uses the fact that an insertion chain can be cut off whenever it produces an insertion expression that can also be derived using just the hypothesis rows of previous mappings in this chain (which describe tuples that already exist in the database). The argument for showing non-termination of a chain goes along the following lines. Let E' be an insertion expression created by mapping σ', generated by pushing E down the template, i.e.,

$$\exists\ w_k \mid \sigma'\colon w_k \leftarrow E, \text{ and } \sigma'(w_{s+1}) = E'.$$

If E' is not new, then there exists some mapping σ'' such that $\sigma''(w_{s+1}) = E'$, and no row in σ'' is equivalent to E. Mapping σ'' can only be constructed from existing information (i.e., its hypothesis rows must correspond to expressions used in forming previous mappings in the chain). Using substitution pattern properties (Chapters 3 and 4), it will be shown that σ'' exists only in special cases. Hence, E' is usually a new insertion expression, and chains can be shown to have unbounded length.

In particular, for the special class of non-simple templates mentioned previously, it will be shown that, given an insertion expression $\sigma_{i-1}(w_{s+1})$ in a chain, it is always possible to find a row w_j such that a subsequent mapping $\sigma_i\colon w_j \leftarrow \sigma_{i-1}(w_{s+1})$ yields a new insertion expression. Hence, for such non-simple templates, chains can always be extended.

Consider a minimal simple full template dependency defined over a set of attributes X. Let an attribute A be added to this template dependency such that all rows except for one match over A, and let the conclusion row be assigned the symbol that does *not* match the others. Call this an *A-perturbation*. The template that follows exemplifies the A-perturbation introduced to the join dependency *[KL, KM, LM], where X=KLM.

K	L	M	A
k	l	m_1	α_1
k	l_1	m	α_1
k_1	l	m	$\bar{a}$
k	l	m	$\bar{a}$

Let the non-matching symbol - $\bar{a}$ - be introduced in row w_k, and let all remaining rows have the symbol α_1 for attribute A. Consider forcing the insertion of $\{t\}_i$ and the resulting set of insertion chains. Two cases should be considered: first, the initial mapping of a chain is generated by replacing row w_k by $\{t\}_i$; second, this initial mapping is generated by replacing some other row $w_{j\neq k}$ by $\{t\}_i$. When the subsequent mappings in the chain are considered, it is shown that all but one of the substitutions for this template generate new insertion expressions, and thus it is always possible to extend a chain. The analysis of the second case is very similar to that of the first one, and will therefore will be omitted.

I) The initial mapping is generated by σ_1: $w_k \leftarrow \{t\}_i$

The pattern that corresponds to this mapping can be projected into two disjoint sets of attributes: the pattern formed by the original simple template over X, where any insertion chain stops after one iteration; and the added column corresponding to attribute A, where $\bar{a} \leftarrow \Pi_A\{t\}_i$. The new template is still a full dependency, and

$$\Pi_A \sigma_1(w_s+_1) = \Pi_A\{t\}_i .$$

In the picture that follows, the template on the left represents the mapping generated by σ_1: $w_k \leftarrow \{t\}_i$. The next mapping in the chain, σ_2, can have two possible patterns, as far as attribute A is concerned: the template in the middle corresponds to σ_2: $w_k \leftarrow \sigma_1(w_s+_1)$; the one on the right corresponds to σ_2: $w_{l\neq k} \leftarrow \sigma_1(w_s+_1)$.

	X	A	Case 1			
$w_{l\neq k}$	...	α_1	...	α'_1	x_1	A
w_k	X	A	x_1	A	...	$\bar{a}'$
	x_1	A	x_2	A	x_3	$\bar{a}'$

Symbols (x_i) and (...) correspond to expressions that would have been obtained for the original simple template. The chain stops if $\sigma_2(w_s+_1)$ is not new, i.e., it can be generated without using $\sigma_1(w_s+_1)$, and using only the hypothesis rows of σ_1 and σ_2.

Case 1: σ_2 is generated by σ_2: $w_k \leftarrow \sigma_1(w_s+_1)$.
The conclusion row can be divided into $\Pi_X\sigma_2(w_s+_1)$ and $\Pi_A\sigma_2(w_s+_1)$. For the simple template over X, lemma 3.3 shows that $\Pi_X\sigma_2(w_s+_1)$ is not new, since it can also be obtained in σ_2': $w_k \leftarrow \sigma_1(w_k)$ — i.e., by replacing $\sigma_2(w_k)$ by $\sigma_1(w_k)$. $\Pi_A\sigma_2(w_s+_1) = \Pi_A\{t\}_i$ corresponds to the perturbation. However, since $\sigma_1(w_k) = \{t\}_i$, $\Pi_A\sigma_2(w_s+_1)$ can also be obtained by σ_2': $w_k \leftarrow \sigma_1(w_k) = \{t\}_i$. Thus, $\sigma_1(w_s+_1)$ is equivalent to $\sigma_2(w_s+_1)$ and $\sigma_2(w_s+_1)$ does not represent a new insertion expression. (This is the only situation for which an insertion chain will end, for this type of template dependencies.)

Case 2: σ_2 is generated by σ_2: $w_l \leftarrow \sigma_1(w_{s+1})$, $w_l \neq w_k$.

$\sigma_2(w_{s+1})$ is again considered in two parts. $\Pi_X \sigma_2(w_{s+1})$ can still be obtained by σ_2: $w_l \leftarrow \sigma_1(w_l)$. $\Pi_A \sigma_2(w_{s+1})$ now contains a "new" symbol, $\bar{a}'$. This means that $\sigma_1(w_l)$ *cannot* replace $\sigma_2(w_l)$ without modifying the conclusion row. All hypothesis rows, except w_k, match w_l over A:

$$\forall\ w_{j \neq k}, \Pi_A \sigma_2(w_j) = \Pi_A\{t\}_i \text{ and } \Pi_A \sigma_1(w_j) = \alpha_1 \text{ and } \alpha_1 \neq \Pi_A\{t\}_i$$

In words, even if $\Pi_X \sigma_2(w_{s+1})$ can be obtained by a different substitution, this substitution will introduce new expressions in the hypotheses, because of attribute A, and chain termination cannot be shown.

Only two possibilities exist for combining rows of σ_2 with rows of σ_1 without disturbing the pattern for A:

- Possibility 1: replacing $\sigma_2(w_l)$ by $\{t\}_i = \sigma_1(w_k)$, yielding mapping σ_2';
- Possibility 2: replacing $\sigma_1(w_k)$ by $\sigma_2(w_k)$ yielding mapping σ_1'.

It will be shown that neither possibility yields $\sigma_2(w_{s+1})$, which is thus a new insertion expression.

Possibility 1

Consider first modifying mapping σ_2. Attribute A does not need to be considered any longer, since the replacement indicated (of $\sigma_2(w_l)$ by $\{t\}_i$) satisfies the pattern for A. Since σ_2 was generated by modification of row w_l, then all remaining rows $w_{i \neq l}$ match $\sigma_2(w_l)$ over $\mathcal{S}$ame(i,l), and have new symbols for the other attributes.

Recall that for simple full templates $\mathcal{S}$ame(i,l) $\subseteq$ $\mathcal{S}$ame(l,s+1) and $\mathcal{S}$ame(i,l) $\subseteq$ $\mathcal{S}$ame(i,s+1). The situations for which replacing $\sigma_2(w_l)$ by $\sigma_1(w_k) = \{t\}_i$ will not demand introduction of new hypothesis expressions occur when the symbols in $\mathcal{S}$ame(i,l) are not affected:

1) $\mathcal{S}$ame(l,s+1)$= \varnothing \Rightarrow$ $\mathcal{S}$ame(i,l) $= \varnothing$

No row matches w_l, which means the original template over X was not minimal.

2) $\mathcal{S}$ame(i,l) $\subseteq$ $\mathcal{S}$ame(l,s+1) $\subseteq$ $\mathcal{S}$ame(k,s+1)

Again, the original template was not minimal.

3) $\mathcal{S}$ame(i,l)$=\varnothing$, i.e., w_l only matches w_{s+1}

In this case, the conclusion obtained in σ_2': $w_l \leftarrow \sigma_1(w_k)$ corresponds to a different (new) insertion expression, i.e., $\sigma_2(w_{s+1}) \neq \sigma_2'(w_{s+1})$.

Possibility 2

$\sigma_2(w_k)$ contains the same values as $\sigma_1(w_{s+1})$ for attributes in $\mathcal{S}$ame(k,l) and new symbols for the remaining attributes. Let $\sigma_1(w_k)$ be replaced by $\sigma_2(w_k)$, generating σ_1'. As in Possibility 1, new hypothesis expressions occur only if

- w_k matches no other row (and thus the result is different), or
- w_k is subsumed by w_l (i.e., the embedded simple template was not minimal).

Summarizing the last three pages, consider a minimal simple full template into which an A-perturbation is introduced, and the set of chains generated by forcing insertion of $\{t\}_i$. Let σ_1 be generated by replacing w_k by $\{t\}_i$, where w_k is the row that contains the non-matching symbol $\bar{a}$ that appears in the conclusion. The insertion expression obtained, $\sigma_1(w_{s+1})$, represents a "new" insertion that will generate another (k-1) "new" insertions for any subsequent mappings generated by σ_2: $w_{l \neq k} \leftarrow \sigma_1(w_{s+1})$.

What remains to be shown is that some of these mappings σ_2 will again originate new insertion expressions, and that at each step chains can be extended. The argument is the same as the one employed in the transition from σ_1 to σ_2 — (k-1) substitutions generate new insertion expressions — and is based on the fact that the transition between any two mappings σ_{i-1} and σ_i is characterized by either

○ a) A "new" symbol for attribute A is introduced in the result $\sigma_i(w_{s+1})$, whenever σ_i is generated by σ_i: $w_{l \neq k} \leftarrow \sigma_{i-1}(w_{s+1})$, or

○ b) The result contains an "old" symbol for A carried over from mapping σ_{i-1}, when σ_i: $w_k \leftarrow \sigma_{i-1}(w_{s+1})$. In this case all other rows $\sigma_i(w_{l \neq k})$ will have acquired a "new" symbol for attribute A.

Following the lines of the previous discussion, it can be shown that
$\sigma_i(w_{s+1})$, in case (○ a), always describes a new insertion; and
$\sigma_i(w_{s+1})$, in case (○ b), is equivalent to $\sigma_{i-1}(w_{s+1})$.

Thus, for an A-perturbation to a minimal simple template, an insertion chain terminates only if a substitution is performed on row w_k.

II) Generalizing by partitioning over A

The previous results can be generalized to template dependencies defined over two disjoint sets of attributes, X and Y, such that for attributes in X it is a minimal simple dependency. Each column y $\in$ Y is characterized by having two repeated symbols.

Example: Consider the dependency

Y_1	Y_2	Y_3
y_1	y_2	y''
y_1	y'	y_3
$\bar{y}$	y_2	y_3
$\bar{y}$	y'	y''

The request $(In, \{t\}_i, R_i, force)$, for this template, will generate unbounded insertion chains. This was the template used for the example at the end of Chapter 3, where $(Y_1 Y_2 Y_3)$ = (ABC). Notice that the set of attributes corresponding to the embedded minimal simple dependency is empty.

Consider now generalizing the problem of A-perturbations so that several rows are allowed to match over A=α_1, and others to match over A=$\bar{a}$. Assume, again, that the (original) simple template dependency over X is minimal.

This generalized problem can be treated by

a) Dividing the template vertically (i.e., by "projection") into two sets of attributes: one corresponds to the minimal simple template dependency and the other to the perturbation introduced; and

b) Dividing the template horizontally (i.e., by "selection") into two sets of rows: each has the same symbol for attribute A.

1	(I)	-	α_1 (II)
2	(III)	-	$\bar{a}$ (IV)
w_{s+1}		...	$\bar{a}$

For generation of new symbols, quadrants (II) and (IV) are treated as distinct units, and quadrants (I) and (III) correspond to the simple dependency. Let $w_{(1)}$ denote any row in the top horizontal region (1), and $w_{(2)}$ any row in the bottom horizontal region (2).

Notice that the bottom region by itself represents a simple dependency. Thus, by lemma 3.3, for any rows $w_{(2)}$ and $w'_{(2)}$, not necessarily distinct, the mappings σ_1: $w_{(2)} \leftarrow \{t\}_i$ and σ_2:$w'_{(2)} \leftarrow \sigma_1(w_{s+1})$ are such that $\sigma_2(w_{s+1})$ does not represent a new insertion expression. In other words, if any chain contains two successive replacements in the bottom region, the second replacement does not yield a new insertion expression.

Furthermore, if the top region consists of a single row, then the modified template dependency still corresponds to a simple template dependency. Assume, therefore, that the top region (1) has at least two rows. New insertion expressions will be obtained as long as the mappings generated are of the form

$$\sigma_{i-1}\text{: } w_{(1)} \leftarrow E \text{ and } \sigma_i\text{: } w'_{(1)} \leftarrow \sigma_{i-1}(w_{s+1}).$$

The reason for this lies in the symbol generation pattern for quadrants (II) and (IV). The picture shows two such successive mappings:

$$\begin{array}{ll|lcl|l}
(1) & \ldots & \bar{a}_k\ (E) & & \ldots & \bar{a}_i \\
 & \ldots & \bar{a}_k & & xx & \bar{a}_i\ (E') \\ \cline{3-3}\cline{6-6}
(2) & \ldots & \bar{a}_i & \rightarrow & \ldots & \bar{a}_{i+1} \\ \cline{2-3}\cline{5-6}
w_{s+1} & xx & \bar{a}_i\ (E') & & \ldots & \bar{a}_{i+1}
\end{array}$$

No row in region (1) of mapping σ_{i-1} can replace the row in region (1) that originates the subsequent mapping σ_i without disturbing the mapping pattern. The minimality of the template over X precludes the replacement of any row $w_{(1)}$ in σ_i by some row $w_{(2)}$ in σ_{i-1}.

Consider now another type of mapping sequence,
σ_{i-1}: $w_{(1)} \leftarrow E$ and σ_i: $w_{(2)} \leftarrow \sigma_{i-1}(w_{s+1})$

$$\begin{array}{ll|lcl|l}
(1) & \ldots & \bar{a}_k\ (E) & & \ldots & \bar{a}_{k+1} \\ \cline{3-3}\cline{6-6}
(2) & \ldots & \bar{a}_i & \rightarrow & xx & \bar{a}_i\ (E') \\ \cline{2-3}\cline{5-6}
w_{s+1} & xx & \bar{a}_i\ (E') & & & \bar{a}_i
\end{array}$$

This corresponds to considering a simple template formed by the set of rows in (2) and the additional row in region (1) whose replacement yielded E'. In this simple template, the expression $\sigma_i(w_{s+1})$ can be obtained if, in mapping σ_i, the row $w_{(2)}$ whose replacement generated σ_i is replaced by $\sigma_{i-1}(w_{(2)})$. In this case, $\sigma_i(w_{s+1})$ is equivalent to $\sigma_{i-1}(w_{s+1})$. The proof is the same as the one used for lemma 3.3.

Summarizing the results for templates partitioned on attribute A, any chain that ends in region (2) is bounded; chains that remain in section (1) are unbounded. Thus, forcing insertions to preserve this type of non-simple template dependency gives origin to insertion chains of unbounded length.

III) Generalizing to several non-simple columns

Consider, finally, the same type of A-perturbation described in (II) extended to additional attributes. This can again be analyzed by letting the template be divided in horizontal regions.

Example:

	X	A	B	C ...
(1)	...	a	b	c
(2)	...	a	$\bar{b}$	c
(3)	...	a	$\bar{b}$	$\bar{c}$
(4)	...	$\bar{a}$	b	$\bar{c}$
w_s+1	...	$\bar{a}$	b	c

Assume that the attributes in X correspond to a non-empty minimal simple template. The description of the set of finite chains, for each attribute is:

A : any sequence of mappings that remains or ends in section (4)
B : any sequence of mappings that remains or ends in sections (1) or (4)
C : any sequence of mappings that remains or ends in sections (1) or (2)

Since the intersection of these conditions is the empty set, no chain will ever stop for this template. If, instead, the conclusion row contains $(\bar{a}b\bar{c})$, then all chains that remain or end in section (4) are bounded.

6.6. Extending the update validation algorithm to support universal relation databases

Some theoretical research on relational databases is based on the *universal relation model*, in which the database is considered as representing a single relation over a *universal scheme*. Among several approaches to this model, the one that has attracted the most attention is the *weak instance approach*. Instead of a single universal relation, it assumes that there exists a *set* of possible (weak) universal relations, where any relation in the database is contained in the projection of some weak instance.

Universal relation views assume that the database represents a single relation over a universal scheme. As remarked before, the update validation algorithm assumes that the underlying relation schemes are independent and therefore can be updated separately. Thus, it applies to any universal relation scheme composed of independent schemes. If the schemes are not independent, inter-relation constraints must be taken into consideration.

The problem of maintaining inter-relation constraints is aggravated by the issue of forced updates. For instance, a forced insertion into a given relation may also require forcing updates (not necessarily insertions) in other relations. Even if the only dependencies allowed in the universal schemes are functional dependencies (which is the update model of Brosda and Vosgens [BRO85]), taking forced insertions into account may generate an infinite chain of pairs $(w_{fd},[w'_{fd}])$. This would not happen, for instance, if the universal scheme itself were in Partitionable BCNF.

If the update validation algorithm were to support the universal instance assumption for general constraints, it would need to check sets of constraints over sets of relations. This, in its turn, would imply templates spanning relation schemes, to be checked concomitantly with the templates for each individual scheme. The existence of several templates to be chased in order to determine additional updates implies that the chase may never end. Assuming that the dependencies given are such that this will not occur, and that furthermore the dependencies allowed do not generate ambiguous updates or infinite chains, the next problem that appears in considering inter-relation constraints in the update validation algorithm lies in determining appropriate $Op(\{t\}_i, R_i, exception)$ rules. The paragraphs that follow comment on findings of other authors on the complexity of maintaining a universal instance. This leads to the belief that establishing $Op(\{t\}_i, R_i, exception)$ rules for inter-relation constraints is a very hard problem.

Fagin and Vardi [FAG84] point out that queries posed to a (weak) universal relation interface have to be computed over the intersection of all the weak instances. Whereas computing answers with respect to functional dependencies can be done in polynomial time, computing answers with respect to full dependencies is EXPTIME-complete, and for embedded dependencies is unsolvable. Since update processing must maintain query consistency, extending the analysis to general universal relation models may not be possible.

Graham and Mendelzon [GRA82] analyze weak instance dependency satisfaction (see Chapter 2) for tuple generating dependencies and equality generating dependencies, which generalize template dependencies and functional dependencies. They define the concepts of completeness (which in the case of template dependencies for one relation means that all tuples generated by chasing the template must exist in the view); and consistency (which in the case of functional dependencies for one relation means that all tuples in the relation agree over the functional dependencies). They show that, for a single relation, standard satisfaction (of a weak instance) is the conjunction of consistency and completeness. Thus, the $Op(\{t\}_i, R_i, exception)$ rules are in fact a way of achieving standard satisfaction, and the updated relation is complete and consistent under their definition.

However, if sets of constraints are imposed on sets of relations, consistency and completeness of a state are reducible to implication problems for dependencies. As the authors point out, this means that it is not probable that consistency can be checked efficiently, except for very restricted cases of dependencies (e.g., functional dependencies or multivalued dependencies). Thus, ensuring maintenance of consistency after an update is performed is also bound to be intractable.

Finally, they point out that, for some cases of universal relations, the dependencies can be projected into each relation, and then only local consistency needs to be maintained. This is the type of situation handled by the update validation algorithm, assuming the view designer has already determined the projected dependencies. Again, projected dependencies can be determined for special cases, but this determination is computationally hard.

6.7. Other extensions

The update validation algorithm should also be extended to include other types of view generating functions, as well as other integrity constraints.

Other operators for the q_v function should, in particular, include set union and difference. The main problem with difference is that it is not monotonic: a deletion in a relation may cause insertion of tuples in the view. Therefore, views formed by difference operators would have to be processed in a special way. The $Op(\{t\}_i, R_i, exception)$ rules would still be valid, since they are applied before execution of the generating function q_v, but the interpretation of intermediate results, database state descriptions and the final rows would have to change. Since difference is not a monotonic operator, most of the theorems — which were proved correct for SJP views — would not hold any longer. It is for example conceivable that isolated execution of underlying updates (Theorem 10) would no longer be sufficient for deriving all possible side effects. This would result into an exponential execution time for the update validation algorithm process.

The union operator (which is monotonic) can be tentatively implemented by translating it into the union of tableau rows in the sequence $T_1 \cdots T_k$. This again presents a problem for interpretation of the output: for instance, deleting from a relation does not necessarily imply elimination of the tuple from the union. Furthermore, there would have to be changes in the manipulation of database state descriptions.

Other types of constraints should also be analyzed, such as those described by nontyped template dependencies. This would allow, for instance, the treatment of inclusion dependencies and referential integrity constraints, which represent yet another type of inter-relation constraints. The first problem that occurs is that chasing with inclusion dependencies may not be finite (and thus forcing insertions given inclusion dependencies may generate yet another modality of infinite chains). Even when restricted to individual relations, untyped dependencies present the problem of adequate $Op(\{t\}_i, R_i, exception)$ rules. If R_1=(AB) is subject to the inclusion dependency A$\subseteq$ B, deletion of (aB) may require deletion of (Ab) as well, and insertion of (AB) may not be allowed.

An initial insight into the problem of untyped dependencies may be gained from the analysis of Minker and Nicolas [MIN83]. Their class of *singular* recursive axioms, whose derivation paths are finite, includes a special subset of untyped template dependencies. This subset corresponds to template dependencies with Nh_i hypothesis rows, of which Nh_i-1, together with the conclusion row, characterize a simple template dependency. The remaining row consists of a permutation of the symbols of the conclusion.

6.8. Conclusions

The update validation algorithm as a design tool

The update validation algorithm should be used as a means of detecting cost-effective translation policies, and those that will most likely result in minimality of changes. If the designer is allowed to associate different translations with different view updates, the set of side effects resulting from each individual policy provides an indicator for choosing a more adequate update translation.

Prototype models of the database usage could be designed, from which information about data modification profiles would be inferred. Since the update validation algorithm associates the possible view changes with the database states for which they occur, this would make it possible to determine which update mappings would cause the least amount of changes (given the most common database modifications). The fact that the policies allowed here accept coexistence of insertions, deletions and changes may even show that it is sometimes cheaper to implement a deletion by a forced insertion.

Other activities which would profit from the diagnostic provided by the update validation algorithm include detection of update anomalies when integrating new views into an existing database, and when merging existing applications to form a new system.

Some other open topics in the view update problem

Until now, views have been assumed to be single-relation answers to queries, but it might be sometimes more adequate to have multiple relation views. The specification and mapping of updating through these views has yet to be analyzed.

Another issue is that of finding appropriate update mappings. This thesis assumes the user has an idea of reasonable mappings to use. There is, however, a pressing need for better guidelines to defining appropriate update mappings, since theoretical work has until now worried only about general mappings. The abstract data type approach, which is a step in this direction, does not provide any indication of how to choose a "good" translation. Relaxing Keller's [KEL85] assumptions about systematic update translation algorithms would probably provide some insight into this area.

Forcing of updates, as suggested in [FAG83], should sometimes be allowed to change the integrity constraints themselves. This should also consider the treatment of exceptions, e.g., when a clerical error assigns the same license plate number to two different people. (While characterizing violation of a functional dependency, this may not be discovered until several transactions have been processed for each individual. Arbitrarily changing one of the license plates is unlikely to be the best solution.)

Finally, very little has been done about dynamic modelling of updates. If the abstract data type approach is extended into this direction, for instance, it is conceivable that different update translations be specified to be triggered for different initial database states.

Bibliography

[AHO79] Aho,A.V., Sagiv,Y., and Ullman,J.D. "Equivalences among relational expressions." *SIAM Journal of Computing, 8(2), 1979,* 218-246.

[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.

[BIS83] Biskup,J. and Bruggemann,H.H. "Universal relation views: a pragmatic approach." *VLDB 1983,* 172-181.

[BRO85] Brosda,V. and Vossen,G. "Updating a relational database through a universal relation schema interface." *PODS 1985,* 66-75.

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

[CAR80] Carlson,C.R., Arora,A.K. and Carlson,M.M. "The application of data dependency theory to the study of databases." *COMPSAC 1980,* 655-660.

[CAS84] Casanova,M.A. and Furtado,A.L. "Updating relational views." *Relatorio tecnico 020, IBM do Brasil, 1983.*

[CHA83] Chan,E.P.F. and Mendelzon,A.O. "Independent and separable database schemes." *PODS 1983,* 288-296.

[CLE78] Clemons,E.K. "An external schema facility to support database update." in *Databases: improving usability and responsiveness.* B.Shneiderman, editor. Academic Press,N.Y., 1978, 371-398.

[COD70] Codd,E.F. "A relational model of data for large shared data banks." *CACM 13(6), 1970,* 377-387.

[COD79] Codd,E.F. "Extending the database relational model to capture more meaning." *ACM TODS 4(4), 1979,* 397-434.

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

[CRE83] Cremer,A.B. and Domann,G. "AIM - an integrity monitor for the database system INGRES." *VLDB 1983,* 167-173.

[DAT83] Date,C.J. *An introduction to database design, vol. II.* Addison-Wesley, 1983.

[DAY78] Dayal,U. and Bernstein,P.A. "On the updatability of relational views." *VLDB 1978,* 368-376.

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

[DAY82a] Dayal,U. and Bernstein,P.A. "On the updatability of network views - extending relational views theory to the network model." *Information Systems 7(1), 1982,* 29-46.

[ELM80] El-Masri,R.A. "On the design, use and integration of data models." PhD thesis, Department of Computer Science, Stanford University, 1980.

[FAG77] Fagin,R. "The decompositional versus the syntactic approach in database design." *IBM Report RJ2381, 1977.*

[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.

[FAG84] Fagin,R. and Vardi,M.Y. "The theory of data dependencies - an overview." *17th Colloquium in Automata, Languages and Programming, Antwerp, 1984,* 1-22.

[FUR77] Furtado,A.L. and Kerschberg,L. "An algebra of quotient relations." *SIGMOD 1977,* 1-8.

[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.

[GAD82] Gadgill,S.G. and Navathe,S.B. "A methodology for view integration in logical database design." *VLDB 1982,* 142-164.

[GAL84] Gallaire,H., Minker,J. and Nicolas,J.M. "Logic and databases - a deductive approach." *Computing Surveys, 16(2), 1984,* 153-186.

[GRA82] Graham,M.H., and Mendelzon,A.L. "Notions of dependency satisfaction." *PODS 1982,* 177-182.

[HEG83] Hegner,S.J. "Algebraic aspects of relational database decomposition." *PODS 1983,* 400-412.

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

[HON83] Honeyman,P. and Sciore,E. "A new characterization of independence." *SIGMOD 1983,* 92-96.

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

[KEL84] Keller,A.M. and Ullman,J.D. "On complementary and independent mappings on databases." *SIGMOD 1984,* 143-148.

[KEL85] Keller,A. "Updating relational databases through views." PhD Thesis, Department of Computer Science, Stanford University, 1985.

[KLU78] Klug,A.C. "Theory of database mappings." *University of Toronto Technical Report CSRG-98, 1978.*

[KOB84] Kobayashi,I. "Validating database updates." *Information Systems, 9(1), 1984,* 1-7.

[KUP84] Kuper,G.M., Ullman,J.D. and Vardi,M.Y. "On the equivalence of logical databases." *PODS 1984,* 221-228.

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

[MAI83] Maier,D. *The theory of relational databases.* Computer Science Press, Rockville,Md, 1983.

[MAI84] Maier,D., Vardi,M. and Ullman,J.D. "On the foundations of the universal relation model." *TODS 9(2), 1984,* 283-308.

[MAS84] Masunaga,J. "A relational database view update translation mechanism." *VLDB 1984,* 309-320.

[MIN83] Minker,J. and Nicolas,J.M. "On recursive axioms in deductive databases." *Information Systems, 8(1), 1983,* 1-13.

[MOT81] Motro,A. and Buneman,P. "Constructing superviews." *SIGMOD 1981,* 56-64.

[NEU82] Neumann,T. and Hornung,C. "Consistency and transactions in CAD databases." *VLDB 1982,*

[NIC78] Nicolas,J.M. and Yazdanian,K. "Integrity checking in deductive databases" In *Logic and databases*. H.Gallaire and J.Minker, editors. Plenum Press, N.Y. 1978, 325-343.

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

[PAO77] Paolini,P. and Pelagatti,G. "Formal definitions of mappings in a database." *SIGMOD 1977.*

[REI81] Reiter,R. "On the integrity of typed first order databases." *Advances in database theory, vol I.* H.Gallaire, J.Minker, and J.M.Nicolas, editors. Plenum Press, NY., 1981, 137-158.

[SAD80] Sadri,F. and Ullman,J.D. "The interaction between functional dependencies and template dependencies." *SIGMOD 1980,* 45-51.

[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.

[SPY80] Spyratos,N. "Translation structures of relational views." *VLDB 1980,* 411-416.

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

[STO75] Stonebraker,M. "Implementation of integrity constraints and views by query modification. "*SIGMOD 1975,* 65-77.

[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.

[VIA83] Vianu,V. "Dynamic constraints and database evolution." *PODS 1983,* 389-399.

[WON80] Wong,E. and Katz,R.H. "Logical design and schema conversion for relational and DBTG databases. In Entity-Relationship Approach to Systems Analysis and Design, Editor P.P.Chen, North-Holland Publishing Co., 1980, 311-321.