UPDATING RELATIONAL DATABASES THROUGH VIEWS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Arthur Michael Keller
June 1995

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Gio C.M. Wiederhold
(Medicine and Computer Science)
(Principal Advisor)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Jeffrey D. Ullman
(Computer Science)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Christos H. Papadimitriou
(Computer Science and
Operations Research)

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

To Barbara,
Marty,
and Paul

Abstract

We consider the problem of updating databases through views composed of selections, projections, and joins applied to Boyce-Codd Normal Form relations. The updates are expressed against the view and they must be translated into updates expressed against the database. We present and justify five criteria that these translations must satisfy. For each type of view update (insert, delete, replace), we provide a list of templates for translation into database updates that satisfy the five criteria. We show that there cannot be any other translations that satisfy the five criteria. A translator can be chosen at view definition time by asking the definer of the view a series of questions based on the view definition and database schema. We conclude that it is practical to update relational databases through views for a large class of views by choosing a view update translator at view definition time.

Acknowledgements

I would like to thank Gio Wiederhold for his patience, advice, support, friendship, and leadership. Voy Wiederhold is my "local Jewish mother" while I was writing the thesis. Carolyn Tajnai was my "local Jewish mother" before that. I learned about the use of proofs in research from Jeff Ullman. Mario Schkolnik of IBM introduced me to the problem while I was an Academic Associate for him at the San Jose Research Laboratory. Pat Selinger was originally on my thesis committee until our interests diverged; I hope to work with her again in the future. Christos Papadimitriou provided perspective from his work on view updates and complexity. Gio Wiederhold, Jeff Ullman, Mark Linton, Don Knuth, and Walter Murray served on my orals committee.

Brent Hailpern and Jim Celoni, S.J., put up with me for an officemate until I got my own tiny office. Denny Brown taught me many things about lecturing.

This thesis was formatted using the TEX document formatter developed by Donald E. Knuth and maintained by Dave Fuchs using the LATEX macro package developed by Leslie Lamport with enhancements for Stanford theses by Howard Trickey and Gabi Kuper. This thesis was printed on the Imagen 8/300 using software developed by Arthur Samuel and on the Xerox Dover. I extensively used the SAIL computer running the WAITS operating system, which only continues to run through the efforts of Marty Frost and Joe Weening.

Students and staff of the KBMS and System/U projects were very helpful. I benefited greatly from discussions with Jim Davidson. Shel Finkelstein was very helpful during the early stages. I worked on the related problems of incomplete information with Marianne Winslett Wilkins. Since her retirement, I miss the help I used to get from Jayne Pickering. However, Ariadne Johnson has also been of invaluable assistance.

I learned a lot about computing from Howard Givner at Brooklyn College. I took my first computer course thirteen years ago from Sylvia Schwartz at Stuyvesant High School, New York City.

There are many people who made my time at Stanford enjoyable and productive. It would be impossible to list all of them. I shall mention a few of the more important ones: Doug Appelt, Richard Beigel, Joan Feigenbaum, Ruth Ferziger, Sheryl Flieder, John Gilbert, Ralph Gorin, Amy Lansky, Richard Manuck, Peter Rathmann, Betty Scott, Laurie Sinclair, Maureen Tjan, John Van Natta, Dave Wall, and Don Woods.

Some of the research presented here was previously published [Keller 82, 84b, 85a, 85b].

This research could not have been possible without the funding that I received to undertake it. These sources include a National Science Foundation Graduate Fellowship; contract S941478 of the IBM San Jose Research Laboratory; contract 9083403 (S-1 Project, Prof. Gio Wiederhold, Principal Investigator) from Lawrence Livermore Laboratory of the Department of Energy; contracts N39-80-G-0132, N39-82-C-0250, and N39-84-C-0211 (the Knowledge Base Management Systems Project, Prof. Gio Wiederhold, Principal Investigator) from the Defense Advanced Research Projects Agency; grant AFOSR-80-0212 (Universal Relations Project, Prof. Jeff Ullman, Principal Investigator) from the Air Force Office of Scientific Research, both of the United States Department of Defense; grants IST-79-21977 (TEX Project, Prof. Donald Knuth, Principal Investigator) and MCS-82-03404 (Universal Relations Project, Prof. Jeff Ullman, Principal Investigator) from the National Science Foundation; and the Computer Science Dept. And it goes without saying that "The views and conclusions contained in this document are those of the author and should not be interpreted as representative of the official policies of DARPA or the US Government."

Contents

Chapter 1

Overview and Introduction

We will consider the problem of updating relational databases through views.

When a shared database is designed, we consider a variety of requirements. For each class of users, we devise a model database design that satisfies the needs of the class. We then combine these individual model database designs into a global integrated database design, which is then implemented [El Masri 79]. However, the resultant database is much too complex for the average user. We need to provide an interface to the user based on the database design developed for his class. We can define a *view* which maps from the underlying global database to a much more manageable portion that the user is interested in. This view is strongly related to the data model originally defined for the class, but will include material learned during the integration process.

The user can query the database through this view to obtain the data desired. The user's query is composed with the view definition to obtain a query that can be applied to the underlying database. This process is called query modification [Stonebraker 75].

Updates, however, currently need to be specified against the underlying database rather than against the view. This considerably reduces the effectiveness of views, as they can be used only for queries.

Since in our model, the view is only an uninstantiated window onto the database [Chamberlin 75], any updates specified against the database view must be translated into updates against the underlying database.[1] The updated database state then

[1]This is the classical notion of a database view. If the view is an instantiated relation, we call it

induces a new view state, and it is desired that the new view state corresponds to performing the user-specified update directly on the original view state, were that possible. This is described by the following diagram.

$$\begin{tikzcd}
V(DB) \arrow[r, "U", ""{name=A, below}] & U(V(DB)) \overset{?}{=} V(DB') \\
\text{DB} \arrow[u, "V"'] \arrow[r, "T(U)", ""{name=B}] & T(U)(DB) = DB' \arrow[u, "V"']
\arrow[Rightarrow, from=A, to=B, "T"]
\end{tikzcd}$$

The user specifies update U against the view of the database, $V(DB)$. The view update translator T supplies the database update $T(U)$, which results in DB' when applied to the database. The new view state is $V(DB')$. This translation has *no side effects in the view* if $V(DB') = U(V(DB))$, that is, if the view has changed precisely in accordance with the user's request.

Given a view definition, the question of choosing a view update translator arises. This requires understanding the ways in which individual view update requests may be satisfied by database updates. Any particular view update request may result in a view state that does not correspond to any database state. Such a view update request may not be translated without relaxing the constraint which precludes view side effects. Otherwise, the update request is rejected by the view update translator. If we are lucky, there will be precisely one way to perform the database update that results in the desired view update. Since the view is many-to-one, the new view state may correspond to many database states. Of these database states, we would like to choose one that is "as close as possible," under some measure, to the original database state. That is, we would like to minimize the effect of the view update on the database.

1.1 View Update Translation

We need to define a few terms to explain the process of translation of view updates into database updates [Ullman 82a, Maier 83]. A domain is a (finite) set. A relation schema is an ordered (or tagged) set of domains and a set of constraints that tuples in the relation must satisfy. A functional dependency or key dependency is an example

a snapshot instead.

of such a constraint. A tuple is an ordered (or tagged) set of values, each one from its respective domain. The extension of a relation is the set of tuples in the relation. A database schema is a set of relation schemata indexed by relation name. A database extension is a set of relation extensions, one for each relation in the database schema.

A database view definition is a mapping whose domain is the set of all relation extensions for a given database schema. The range of a database view definition is also a set of relation extensions for a schema specific to the view definition. The mapping from the domain database to each relation in the range of the view is defined by a type of database query. The view extension is the extension of the database which is the range of the view for a particular extension of the database which is the domain of the view.

The primitive update operations on databases and views are deletion, insertion, and replacement. A deletion is the removal of a single tuple from a relation. An insertion is the addition of a single tuple into a relation. A replacement is the combination of a deletion and an insertion into the same relation performed as a single atomic action so that it does not require an intermediate consistent state between the deletion and insertion steps. An update is a deletion, an insertion, or a replacement.

A database update may be directly applied against the database, provided it satisfies the constraints on the database. A view update is merely an update that is described against the view, but it must be translated into a sequence of database updates in order for it to be executed. There may be several candidate sequences of database updates corresponding to one view update. We call these sequences of database updates the *translations* of the view update request.

We say that a translation is *valid* if it performs the view update as requested. For updates through select and project views, we will require that the new view extension be precisely the result of performing the view update on the old view extension, were the view to be an ordinary relation. For updates through views that include joins, it may not be possible to perform the view update without additional changes to the view. These *view side effects* are as a result of functional dependencies that require that changes in the view tuples requested are consistent with the remainder of the database.

Requiring that a translation be valid is not sufficient for our purposes—it is only a first step. We define 5 additional criteria (in Chapter 3) we require the translations to satisfy. We use the criteria to obtain only the simplest (or minimal) view update translations.

A view update translator is a mapping from view update requests into translations of these view update requests. A translator takes the user's view update requests and translates them into database update requests, which will then be processed by the database system. Thus, a view update translation facility is a useful adjunct to a database system which has a view definition facility. The lack of a view update translation facility means that users must specifying updates directly against the database rather than through views. Often users must act indirectly and submit changes through specialists authorized by the database administrator.

We shall provide several (classes of) view update translators. We do not claim that these are the only view update translators possible. Rather, we show that, for each view update request, all possible distinct translations of that view update request into sequences of database update requests that satisfy our 5 criteria are obtained from these view update translators.

We can obtain other view update translators from the translators we define. One translator can be obtained from another by rejecting some updates and translating the others. Given two translators, there are translators that use some translations from translator and some from the other. Using these two translator transformations, from the view update translations we define, we can generate the complete set of view update translators whose translations that satisfy our 5 criteria.

In order that the view update facility may choose among these alternatives at view update time, we propose that the database adminstrator (DBA) provide additional semantics during view definition time. These additional semantics permit the view update facility to choose one of these translators.

With select, project, and join views, we can trace each view tuple to one or more database tuples. We call these database tuples the *corresponding underlying tuples*. The corresponding underlying tuple to a view tuple for a selection view is the database tuple with the matching values. The corresponding underlying tuples for a view tuple for a projection view are the database tuples whose projections match the view tuple. The corresponding underlying tuples for a view tuple for a join view are the database tuples from each relation that join to form the view tuple; they are also projections of the view tuple. For views formed by a combination of selections, projections, and joins, the corresponding underlying tuples are traced individually across the operators from the view down to the database relations through the intermediate views (although these intermediate views may not actually exist).[2]

We have considered single query views in this work. This results in a single relation. For multiple relations presented to the user, multiple view queries can be defined. We call the mixture of individual view queries and database relations presented to the user a *window*. We leave consideration of the differences between queries against windows for future work.

1.2 Motivation

We will now motivate the problem being addressed by showing via a simple example the issues which have to be resolved for effective updating through views.

Let us consider the relation `EMP` which contains each employee's number, name, location, and whether the employee is a member of the company baseball team. The company has two locations: New York and San Francisco. Baseball team members must be employees.

The personnel manager, Susan, in New York has the following view definition:

View P:

```
Select *
From EMP
Where Location="New York"
```

She requests the deletion of employee #17 from her view. A reasonable translation of this request is to delete the employee record from the underlying database. Thus, we have translated a view deletion into a database deletion. If the employee was a member of the baseball team, he has been removed from that membership as well.

The baseball team manager, Frank, has the following view definition:

[2]The notion of corresponding underlying tuples needs the notion of monotonicity, which holds for select, project, and join views.

View B:

```
   Select *
   From EMP
   Where Baseball="Yes"
```

He requests deletion of employee #14 from his view. It is unreasonable to delete the employee tuple from the underlying database (unless you believe that baseball is all-important). A reasonable translation of this view deletion request is to replace the Baseball attribute of the underlying database tuple with a "No." Thus, we have translated a view deletion into a database replacement.

One might argue that the Frank's view deletion request should have been a replacement. However, this would mean requesting the replacement of a tuple in the view with a view tuple that did not appear in the view. Then Frank's request would not be valid in the view, as the replacement tuple could not possibly be a view tuple. In addition, Frank would have to make a distinction between deletion and replacement that he could not discern by looking at the effects through his view.

It is possible to translate the Susan's request by moving employee #17 to California. We doubt that the California manager would be pleased by such an implementation. Rather, such a request should be issued by someone authorized to access the entire relation (as a replacement request); that person would access a view which permits seeing the effects of that request.

We see that a view deletion request is sometimes best translated into a database deletion request and at other times into a database replacement request. As we shall see, similar alternatives arise for insertion and replacement. We suggest that additional semantics be used to choose among the various alternatives, and this will be discussed in Chapter 7.

1.2.1 Motivation for Treatment of Joins

A further level of ambiguity arises when views are constructed using join operations. Consider the following relations, where employee is the key of the ED relation and department is the key of the DM relation, the EDM view is formed by the natural join between the ED and DM relations, and the EM view is also formed by the natural join between the ED and DM relations, but the department attribute is removed by a projection.

ED relation

employee	department
Joe	Art
Sarah	Art
George	Music
Fred	Music

DM relation

department	manager
Art	Sally
Books	Sally
Music	Ira

EDM view

employee	department	manager
Joe	Art	Sally
Sarah	Art	Sally
George	Music	Ira
Fred	Music	Ira

EM view

employee	manager
Joe	Sally
Sarah	Sally
George	Ira
Fred	Ira

Let us consider the request to replace ⟨`George`, `Music`, `Ira`⟩ in view EDM by ⟨`George`, `Music`, `Edo`⟩. This request requires replacing ⟨`Music`, `Ira`⟩ in relation ED with ⟨`Music`, `Edo`⟩. This necessarily has the side effect of changing ⟨`Fred`, `Music`, `Ira`⟩ to ⟨`Fred`, `Music`, `Edo`⟩, as a result of the functional dependency. If we refuse to permit the side effect, we cannot accept the update. The presence of the join attribute in the view update request makes the user's intention clear: the user wishes to change the manager of the Music department for George. It would be reasonable to ask the user to confirm that the intention to also change the manager of other employees in the Music department and in particular, Fred. This affects the user interface and not the algorithms for view update translation, so the confirmation is beyond the scope of this dissertation.

Let us consider some insertion requests and their consequences. Consider the request to insert ⟨`Rita`, `Sally`⟩ into view EM. Is the desired department Art or Books? Consider the request to insert ⟨`Rita`, `Ira`⟩ into view EM. We can assume that the department is Music, but this is not *necessarily* so. Consider the request to insert ⟨`Rita`, `Boris`⟩ into view EM. We have no idea what department to use, since Boris is not already a manager.

Let us consider some replacement requests and their consequences. Consider the request to replace ⟨`George`, `Ira`⟩ in view EM by ⟨`George`, `Sally`⟩. Should George be moved into the Art or the Books department, or should Sally become the manager of Music as well as Art and Books? The latter has the side effect of changing Fred's manager as well. Consider the request to replace ⟨`George`, `Ira`⟩ in the view by ⟨`George`, `Boris`⟩. We could make Boris the manager of the Music department, which would have the side effect of changing Fred's department, or we could move George into a new department and make Boris its manager, but we would have no idea what to call this new department.

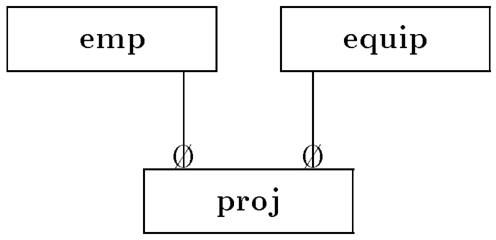
When the join attributes are removed from the view via projections, the view update problem becomes data dependent. Our algorithms are data independent; we decide what to do in advance without consideration of the data, although the data are used to determine whether the update will succeed or fail.

We require that the view have a key which is a key or foreign key of every relation mentioned in the view. This is because deletion from the view is translated as deletion from the "root" relation (this will be explained in detail in Chapter 6).

Consider the following example, which involves a query graph with two roots. We have three relations, *emp*, *proj*, and *equip*, containing information on employees, projects, and equipment, respectively. There is a reference connection from the `proj` attribute of the *emp* relation to the `name` attribute of the *proj* relation, and also a reference connection from the `proj` attribute of the *equip* relation to the `name` attribute of the *proj* relation. The view is defined by the following:

```
DEFINE VIEW user-view AS
SELECT ⟨attributes⟩ FROM emp, proj, equip
WHERE (emp.proj=proj.name) AND (proj.name=equip.proj)
```

This view definition can be described by the query graph (see Chapter 6) in the following diagram. Each node represents a relation, and each directed edge represents a join from the many-to-one direction.



Suppose that we wanted to delete the view tuple asserting that employee Smith from the Parts project has a personal computer at his disposal. We could either delete the record of the employee or of the personal computer. If we delete the Smith tuple in *emp*, not only will the desired view tuple disappear but also all other view tuples mentioning Smith—an undesirable and arbitrary side effect. Similarly, deleting the personal computer tuple in *equip* will cause all tuples mentioning that personal computer to be disappear. Deleting both the employee and equipment tuples could be done (and is suggested by symmetry), but this will cause any view tuple that mentions either Smith or the personal computer to be removed—an even more undesirable side effect. This is why updating views that involve joins not corresponding to a rooted tree should be avoided.

Chapter 2

Previous Work

This chapter is a summary of related published work by others. Our work is most closely related to that of Umeshwar Dayal [78, 79, 82].

2.1 Dayal and Bernstein

Dayal and Bernstein [78, 79, 82] provide some algorithms for translating a restricted class of view updates. They formalize the notion of a correct translation, but it is more restrictive than our five criteria.[1] In particular, they do not permit side effects in the view. The provide some algorithms for translation of view operations into like database operations (e.g., insertions into insertions). They do show how to determine when there is a functionally determining source or clean source, which is a set of tuples that determine the relevant view tuples but do not affect other view tuples. That concept is necessary only when view side effects are proscribed.

Dayal and Bernstein allow select, project, and join views, and do not require join attributes to appear in the view. They generate one translation for each update. We enumerate all possible acceptable translations and show how to choose the desired one. Some of their translations do not satisfy our criteria as they perform unnecessary operations.[2] Since they do not support alternative translations, they do not permit

[1]For example, their notion of the rectangle rule is slightly more restrictive than our related Criterion 1, which eliminates database side effects.

[2]The request to replace ⟨`Newton`, `Physics`, `Aristotle`⟩ in view EDM (of [Dayal 82]) by ⟨`Newton`, `Math`, `Euclid`⟩ will result in deletion of ⟨`Physics`, `Aristotle`⟩ from relation DM.

the semantic distinctions that we used in the baseball team manager example in contrast with the New York manager example.

2.2 Bancilhon and Spyratos

Bancilhon and Spyratos [79, 81] show how the concept of a constant complement is relevant to the view update problem. Two views are *complementary* if given the state of each view, there is a unique corresponding database state. If a complementary view is held constant, there is at most one translation of any given view update. The key phrase is "at most one." In general, some updates that could be translated reasonably do not preserve any complement [Keller 85b]. We will discuss complements further in Chapter 4.

The ambiguity inherent in the view update problem is partially captured by the existence of multiple minimal complements. For each complement, there is a unique view update translator. Bancilhon and Spyratos also prohibit view side effects and their work cannot be extended to handle view side effects. Given a view, they do not show how the set of minimal complements can be enumerated. And they do not show how to construct the view update translator given the view and a complement. While this body of work is inspirational, it is far from practical.

2.3 Cosmadakis and Papadimitriou

Cosmadakis and Papadimitriou [84] base their work on Bancilhon and Spyratos [81]. Their work primarily involves analysis and not general algorithms for view update translation. They show how difficult it is to use complements. For example, they show that finding a minimal complement of a given view is NP-complete. They adopt a universal relation assumption [Ullman 82b]; their views are essentially projections of a given relation. The implication of their work is that an approach different than constant complements is needed.

2.4 Davidson

Jim Davidson [81, 83] addresses the related problem of updating databases through natural language. He generates dynamically defined views based on user interaction. He handles project and join views. His joins are more restrictive than ours, as he only allows chains and we allow trees; however, his projections are less restrictive than ours as he does not require join attributes to appear in the view, but this introduces data dependencies.

His approach involves generating alternatives on the fly and choosing among them by various heuristics. The list of alternatives he generates is data dependent; consequently, the translation he chooses is data dependent. The following example illustrates the problem. Consider the following relations, where employee is the key of the ED relation and department is the key of the DM relation, and them EM view is formed by the natural join between the ED and DM relations with department removed by a projection.[3]

```
ED relation             DM relation
employee  department    department  manager
Joe       Art           Art         Sally
Sarah     Art           Books       Sally
George    Music         Music       Ira
Fred      Music         Sports      Keith
Sam       Sports

EM view
employee  manager
Joe       Sally
Sarah     Sally
George    Ira
Fred      Ira
Sam       Keith
```

The request to change George's manager from Ira to Keith results in changing George's department from Music to Sports. The alternative of changing the manager of the Music department from Ira to Keith is rejected because of the potential for

[3]Note that our work does not handle this case because the join attribute has been removed.

side effects. The request to change George's manager from Ira to Suzanne results in changing the manager of the Music department from Ira to Suzanne. This method is chosen even though it has the side effect of changing Fred's manager from Ira to Suzanne because Suzanne is not already a manager, so we cannot move George to Suzanne's department (she does not have one yet).

Davidson's heuristics appear useful for extending our work to handle universal relations. We propose that future work consider an algorithmic approach that does not depend on the database state which is motivated by some of the heuristics Davidson uses.

2.5 Others

Carlson [79] ignored replacement requests by claiming that they can be decomposed into a sequence of insertions and deletions. We believe that replacement requests are different as they do not require a consistent database state between the deletions and insertions. Furtado [79] proposed handling the union operator for defining views and claimed that insertions, deletions, and replacements specified against a view defined by a union should be applied to each of the underlying relations merged. The union operator is useful for distributed databases where a relation is partitioned over several sites. In this case, an insertion must occur at only one site and not, as Furtado suggests, all sites.

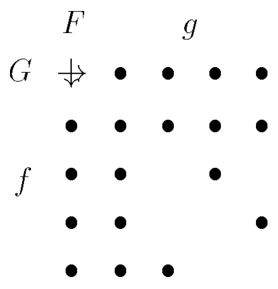
Chapter 3

Complementary and Independent Views

One way to express a limitation of effects of view updates on the database is through the concept of *constant complements* [Bancilhon 81]. Two views are complementary if given the state of each view, there is a unique corresponding database state. Intuitively, this means that the two views are sufficient to reconstruct the database. Bancilhon and Spyratos have observed that by choosing a complementary view and holding it constant, that there is at most one way to translate any update on the given view. They have also shown that if a view is not empty or the identity, then it has multiple minimal complements. A complement is *minimal* if no view providing less information is also complementary. Providing more information does not adversely affect complementarity; therefore, the issue is only interesting when we consider minimal complements.

We observe that choosing a constant complement may cause the view update translator to reject requests that have translations, although none of those translations keep the complement constant. We define two views as independent when any pair of view states corresponds to a database state. When independent views are complementary, it is always possible to hold the state of one view constant while generating any possible state of the other view. Thus, choosing an independent complement, if one exists, permits all updates expressed against the view to be translated to updates expressed against the database. The question then arises whether a view has multiple independent complements. To answer this question, we define a view as *monotonic* if it preserves inclusion (recall that the domain and range of a view is a finite power set). Informally, a view is monotonic if adding tuples to the database does not remove any tuples from the view, although it could augment the view. There are non-monotonic views that have multiple independent complements. However, a monotonic view has at most one complement that is independent and monotonic.

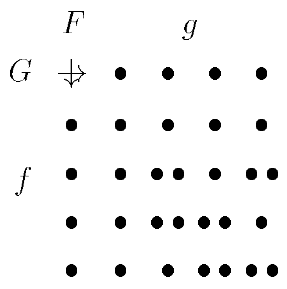
The following diagram illustrates two database mappings f and g.



Each • represents one database state. The domain D is represented by the •'s located in the main quadrant of the diagram (the lower right from the double lines). The function f maps database states of D into database states in F (the left column) by moving across the row. Similarly, the function g maps database states of D into database states in G (the top row) by moving up along the column. The equivalence classes of D induced by f (D/f) are the rows of D. And the equivalence classes of D induced by g (D/g) are the columns of D. We can take the intersection of an equivalence class of D/f and one from D/g; this is represented by one box of the diagram. The two mappings f and g are complementary if, given a database state in F and one in G, there is at most one database state in D that maps into both database states (by their respective mappings). That means each box of the diagram has at most one • in it. This particular diagram illustrates two complementary functions.

Updating the database D through the user view f involves changing view states from some database in F to another (in F). If the mapping g is to be held constant, then the new database state is found by moving along the same column (for the same image in G) from the original row to the desired row. If the mapping to be held constant g is complementary to f, then there is at most one candidate resultant database in D that maps to the same database in G (by g) and maps to the new database in F (by f). When the box (intersection of the view and complement equivalence classes) is empty, the view update request cannot be performed exactly while preserving the complement—the request is rejected; when the intersection contains exactly one database, the view update request has a unique translation that preserves the complement.

The following diagram illustrates two independent functions. Each box in D in the diagram contains at least one •.



Again, updating the database D through the user view f involves changing view states from some database in F to another (in F). If the mapping g is to be held constant, then the new database state is found by moving along the same column (for the same image in G) from the original row to the desired row. If the mapping to be held constant g is independent of f, then there is at least one candidate resultant database in D that maps to the same database in G (by g) and maps to the new database in F (by f). When the box (intersection of the view and complement equivalence classes) contains exactly one database, the view update request has a unique translation that preserves the complement; when the intersection contains more than one database, the view update request has several candidate translations that preserve the complement—the request is ambiguous.

Holding a view complement constant means that whenever the view update is translatable, that translation is unique. If the view held constant is independent (from the user view), then all view update requests can be translated, perhaps ambiguously. If the view that is held constant is both complementary to and independent of the user view, then all view update requests are unambiguously translatable into database update requests. When the views are constrained to be monotonic, if a view has a independent complement, it is unique.

We will now proceed to a formal treatment of the results we have stated informally.

3.1 Definitions

It is assumed that the reader is familiar with database theory [Ullman 82a] and set theory [Halmos 60].

For the purpose of this exposition, a database is a finite power set.

Definition. Let f and g be two functions whose domain is D. (Here we are not concerned with the range of f and g, but only with the equivalence classes induced on D by f and g.) Then f and g are *independent mappings* if

$$[\check{}x, y][((\breve{}d\Delta 1 \in D)(f(d\Delta 1) = x))\Xi((\breve{}d\Delta 2 \in D)(g(d\Delta 2) = y))\text{ß} \\ ((\breve{}d \in D)(f(d) = x\Xi g(d) = y))].$$

The notion of independence we use here is different from Rissanen's notion of independence [Rissanen 77]. His notion stated that two components were independent when the original database could be obtained from them by lossless joins that preserved all dependencies. Our definition relates to the ability to change the selected range value of one mapping while keeping the range value of the other mapping constant. This definition is useful for the problem of view updates, where it is important to consider whether an update specified through a view may be done without affecting another view.

Definition [Bancilhon 81]. Let f and g be two functions whose domain is D. Then f and g are *complementary mappings* if

$$[\check{}x, y \in D][(x\text{æ}y)\Xi f(x) = f(y)\text{ß}g(x)\text{æ}g(y)].$$

Definition. Two functions (mappings) f and g with the domain D are *equivalent* if they induce the same equivalence class on D. (That is $D/f = D/g$. Recall that D/f is defined as follows: $\check{}d \in D, \check{}d' \in D$, d and d' are in the same member of D/f iff $f(d) = f(d')$.)

We observe that independence and complementarity are different properties. Independence means that function can generate all values of its range while another function has a specific range value. Complementarity means that there is at most one element of the domain that simultaneously results in any pair of range values, one from each of two functions. We can give another characterization of these two concepts. Each function f generates a set of equivalence classes D/f. Given two functions f and g we can take the intersection of equivalence classes of D/f with equivalence classes of D/g. If all of these intersections have at most one (domain) element in them, the two functions are complementary. If all of these intersections have at least one (domain) element in them, the two functions are independent.

Definition. Let f and g be complementary and independent functions whose domain is D, and let h be an arbitrary function whose domain is also D. Let the range of f be F and the range of g is G. Since f and g are complementary and independent, there is a one-to-one correspondence between $F \times G$ and D; that is, $d \in D$ corresponds to $a \times b$ where $a = f(d)$ and $b = g(d)$. Then the *coordinatization* of h over f and g is the function h' whose domain is $F \times G$ such that $h(d) = h'(f(d), g(d))$. (We note that h is equivalent to some $h\Delta 0$ iff h' is.)

One question is when is there a unique (up to equivalence) complementary and independent function g for a function f. For example, let $f(x, y) = x$ and $g(x, y) = y$. It is clear that these are independent and complementary. The function $g'(x, y) = 2y$ is independent and complementary to f but also equivalent to g. However, the function $g''(x, y) = x + y$ is independent and complementary to f but not equivalent to g. Since our domain of interest is relational databases, and the mappings of interest are relational views consisting of combinations of select, project, join, and union, we will use a property of these mappings.

3.2 Monotonic Functions

Definition. An n-ary function f whose domain is a finite power set is *monotonic* if $(\breve{}i)(R\Delta i \subseteq S\Delta i)ßf(R\Delta 1, \ldots, R\Delta n) \subseteq f(S\Delta 1, \ldots, S\Delta n)$. (Select, project, join, and union of relations are monotonic functions. The set difference operator, however, is not monotonic.)[1] The composition of monotonic functions is monotonic.

We will now explore some features of complementary, independent, and monotonic functions. We shall require not only that the domain of each function is a finite power set but also that the range be a finite power set as well. This is reasonable since the select, project, and join operators all result in power sets. Let f and g be

[1]The main property of finite power sets we use is that they are complete lattices [Birkhoff 67].

complementary, independent, and monotonic functions whose domain is D and ranges are F and G (all finite power sets), respectively. Since f and g are complementary and independent, there is a one-to-one correspondence between G and the elements of any equivalence classes generated by f. Let G' be the equivalence class of f that contains the empty relation. Using the one-to-one correspondence, we can define $g' : D\text{ß}G'$ as $g'(d) = b$ where $g(b) = g(d)$ and $f(b) = \emptyset$. (Essentially, we have chosen from each equivalence class generated by g the element that maps to the empty set by f.) Similarly, $f' : D\text{ß}F'$ is defined by $f'(d) = a$ where $f(a) = f(d)$ and $g(a) = \emptyset$. Let us define the coordinatization function $c : F' \times G'\text{ß}D$ as $c(f'(d), g'(d)) = d$. Since f and g are complementary, c is a function. Since f and g are independent, c is total. We will now explore some properties of c. First, $c(a, \emptyset) = a$ (and similarly, $c(\emptyset, b) = b$) since $a \in F'$ implies $f'(a) = a$ and $g'(a) = \emptyset$.

Lemma 3.1 *Let $f : F'\text{ß}F$ be a monotonic function. Then $f(\emptyset) = \emptyset$.*

Proof: Let $f(\emptyset) = a$, and let $f(d) = \emptyset$. (Both F and F' must contain the empty set.) Since $\emptyset \subseteq d$ and f is monotonic, $a \subseteq \emptyset$. Therefore, $a = \emptyset$. ∎

Lemma 3.2 *F' is closed under containment.*

Proof: Let $a\Delta1$ be a member of F' and $a\Delta2$ be a subset of $a\Delta1$. Now $g(a\Delta1) = \emptyset$ since $a\Delta1 \in F'$. Since g is monotonic, $g(a\Delta2) = \emptyset$. Therefore, $a\Delta2$ is in F'. ∎

Corollary 3.1 *F' is closed under intersection.*

Lemma 3.3 *Let $f : F'\text{ß}F$ be a monotonic function. If $d \in F'$ and $f(d) = \{e\}$, a singleton, then d is a singleton.*

Proof: Suppose not. Then there exist $d\Delta1, d\Delta2 \in d$ such that $d\Delta1\text{æ}d\Delta2$. By our previous lemma, both $\{d\Delta1\}$ and $\{d\Delta2\}$ are elements of F'. Let $f(\{d\Delta1\}) = a\Delta1$ and $f(\{d\Delta2\}) = a\Delta2$. We note that $a\Delta1\text{æ}a\Delta2$ since f is bijective on $F'\text{ß}F$. But the monotonicity of f implies that $a\Delta1\text{´}a\Delta2 \subseteq \{e\}$. Therefore, $\{e\}$ is not a singleton. ∎

Lemma 3.4 *Let $f : F'\text{ß}F$ be a monotonic function. If $d \in F'$ and $f(d) = a$, then $|d|\,\text{ø}\,|a|$.*

Proof: Let S be the power set of d. Since F' is closed under containment, $S \subseteq F'$. For all s in S, $f(s)$ is in F and $f(s) \subseteq a$. Furthermore, since f is bijective on F'ßF, all the $f(s)$'s are distinct. Then a has as many subsets as d, so a must be at least as big as d. ∎

Lemma 3.5 *Let F and G be finite power sets. Then F' (and also G') is a finite power set.*

Proof: Since F and G are finite power sets, their cardinalities are powers of two. Let $|F| = 2^m$ and $|G| = 2^n$. Then F (G) contains exactly m (n) singletons. Since there is a one-to-one correspondence between D and $F \times G$, $|D| = 2^{m+n}$. Because D is a finite power set, D contains exactly $m + n$ singletons. Let $S\Delta F$ $(S\Delta G)$ be the singletons in F' (G') that map into singletons in F (G). We note that $|S\Delta F| = m$ and $|S\Delta G| = n$ since each singleton of F is mapped into by a unique singleton of F' (and consequently $S\Delta F$). Since f : F'ßF is a bijection, $|F'| = 2^m$. Now, suppose that F' is not the power set generated by $S\Delta F$. Since $|S\Delta F| = m$, the power set of $S\Delta F$ is of size 2^m. As F' is the same size as the power set generated by $S\Delta F$ and they are unequal, there must be some element a in F' that is not in the power set of $S\Delta F$. That set a in F' is then not the union of some singletons in $S\Delta F$. Then there is some element $a\Delta 0 \in a$ not in $S\Delta F$, and $\{\, a\Delta 0 \,\}$ is a singleton that is not in $S\Delta F$. Therefore, F' has more than m singletons. Let us now show that F'\`$G' = \{\, \emptyset \,\}$. Everything in F' is mapped to $\emptyset$ by g, while the only set in G' mapped to $\emptyset$ by g is $\emptyset$. Therefore, the singletons in F' and G' are disjoint. But F' has more than m singletons while G' has at least n singletons. This contradicts the fact that D has precisely $m + n$ singletons. ∎

Lemma 3.6 *Let F, F', G, and G' be finite power sets, and let f : F'ßF be a monotonic function. If $d \in F'$ and $f(d) = a$, then $|d| = |a|$.*

Proof: We have already shown that $|d|$ ø $|a|$. Let $s\Delta i = \{\, s \in F' \mid |s| = i \,\}$. Let $t\Delta i = \{\, t \in F \mid |t| = i \,\}$. Since F and F' are power sets of size 2^n, $|s\Delta i| = |t\Delta i| = \binom{n}{i}$. By induction on $|d|$ we will show that $|d| = |a|$. For $|d| = 1$, $s\Delta 1$ is the set of singletons in F'. Since $|d|$ ø $|a|$, only singletons (elements of $s\Delta 1$) may map into the $t\Delta 1$. But since $|s\Delta 1| = |t\Delta 1| = \binom{n}{i}$, all elements of $s\Delta 1$ must map into elements of $t\Delta 1$. For the induction step, assume that all elements of $s\Delta i$ map into elements of $t\Delta i$ for $1øi < jøn$. We will show that all elements of $s\Delta j$ map into elements of $t\Delta j$. Since $|d|\,ø\,|a|$, the elements of F' that map into elements of $t\Delta j$ must be elements of $s\Delta i$ for $1øiøj$. But by the induction hypothesis, none of these can be elements of $s\Delta i$ for $1øi < j$. Therefore only elements of $s\Delta j$ can map into elements of $t\Delta j$. But $|s\Delta j| = |t\Delta j| = \binom{n}{j}$. Therefore all elements of $s\Delta j$ map into elements of $t\Delta j$. ∎

Theorem 3.1 *Let the domain D be a finite power set. Then F' and F are isomorphic (preserving monotonicity) under f.*

Proof: The preceding lemma ($|d| = |a|$) showed that the singletons of F and D are in one-to-one correspondence. Let $d = \{\, a\Delta 1, \ldots, a\Delta k \,\}$. We will show that $f(\{\, a\Delta 1, \ldots, a\Delta k \,\}) = \{\, f(a\Delta 1), \ldots, f(a\Delta k) \,\}$. Suppose not. Then there is some $f(a\Delta i)$ $(1øiøk)$, say $f(a\Delta 1)$, not in $f(d)$. (From $|d| = |f(d)|$, we know that some of the $a\Delta i$ $(1øiøk)$ are missing and there are other singletons added.) By definition, $\{\, a\Delta 1 \,\} \subseteq d$. But since f is monotonic, this implies $f(\{\, a\Delta 1 \,\}) \subseteq f(d)$. ∎

Corollary 3.2 *f' is monotonic.*

3.3 Conversion to Coordinates

Lemma 3.7 *Let the domain D be a finite power set, and let f and g be monotonic, independent, and complementary. Then all the singletons of D are members of F' or G'.*

Proof: Let 2^m (2^n) be the size of F (G). Then $|F'| = 2^m$ and $|G'| = 2^n$. Since F' (G') is a power set, there are m (n) singletons in F' (G'). Also, $|D| = 2^{m+n}$. Since D is a power set, there are $m + n$ singletons in D. We know that $F' ` G' = \{\, \emptyset \,\}$. Then all of the singletons of D are members of F' or G'. ∎

Corollary 3.3 *For all $d \in D$, there exist $a \in F'$ and $b \in G'$ such that $d = a\,´\,b$.*

Theorem 3.2 *Let the domain D be a finite power set. For c as the function for converting to coordinates defined above, $c(a, b) = a\,´\,b$. (That is, f and g are monotonic, independent, and complementary.)*

Proof: We observe that the theorem holds of $c(a,\emptyset) = a$ and $c(\emptyset, b) = b$. We measure a counterexample $c(a,b) = a'\acute{}b'$ where $a\text{æ}a'$ or $b\text{æ}b'$ and $a' \in F'$ and $b' \in G'$ by the sum of the cardinalities of a and b ($|a| + |b|$). We perform an induction on this measure. Assume that $c(a,b) = a\acute{}b$ for all a and b such that $|a| + |b| < n$. Let $c(a,b) = a'\acute{}b'$ with $a\text{æ}a'$ or $b\text{æ}b'$ and $|a| + |b| = n$. It is not possible that $a' \subseteq a$ and $b' \subseteq b$. (Otherwise either $a'\acute{}b' = a\acute{}b$—assumed false—or $|a'\acute{}b'| < |a\acute{}b|$, which by our induction hypothesis implies $c(a', b') = a'\acute{}b'$.) Without loss of generality, assume that $a' - a\text{æ}\emptyset$. Let $e \in a' - a$. Then $\{e\} \subseteq a'\acute{}b'$, so $f'(\{e\}) \subseteq f'(a'\acute{}b') = a$. Since $e \in a'$, $\{e\} \in F'$ and $f'(\{e\}) = \{e\}$. Since $e \in a' - a$, $f'e \notin a$, a contradiction. ∎

Let us consider the consequence of the preceding theorem and lemmata. Let the B be the basis set of the domain D. (That is D is the power set of B.) Also, let $B\Delta f$ be the basis set of F' and $B\Delta g$ be the basis set of G'. Then $B = B\Delta f\acute{}B\Delta g$.

Theorem 3.3 *Let the domain D be a finite power set. Conversion to coordinates preserves monotonicity. That is, let f and g be monotonic, complementary, and independent functions with domain D and ranges F and G respectively (all finite power sets), and let h be a function with domain D, and let h' be the conversion to coordinates of h over f and g [that is, $h'(f(d), g(d)) = h(d)$]. Then h is monotonic iff h' is.*

Proof: "If." We note that h is the composition of f, g, and h'. Therefore, if h' is monotonic, then h is also.

"Only if." We note that h' is the composition of h and c. If h is monotonic, then h' is also since c is monotonic. ∎

3.4 Uniqueness of Independent, Complementary Mappings

Our question now becomes when does a mapping have a unique (up to equivalence) monotonic, complementary, independent mapping. For domains of finite sets, such mappings are unique.

Theorem 3.4 *Let f, g, and h be monotonic mappings on a domain D, a finite power set, such that the ranges are all finite power sets, and f and g are independent and complementary, as are f and h. Then g and h are equivalent.*

Proof: Let f', g', and h' be the conversion to coordinates of f, g, and h, respectively, over f and g. Let F be the range of f and G be the range of g. Assume that g and h are not equivalent (and consequently, g' and h'). Then there exists some $a \in F$ and $b\Delta 0 \in G$ such that $g'(a, b\Delta 0) = g'(\emptyset, b\Delta 0)$ but $h'(a, b\Delta 0) \neq h'(\emptyset, b\Delta 0)$. Since conversion to coordinates preserves monotonicity, $h'(\emptyset, b\Delta 0) \subset h'(a, b\Delta 0)$. Choose $b\Delta 1$ such that $h'(a, b\Delta 1) = h'(\emptyset, b\Delta 0)$. (Such a $b\Delta 1$ must exist since f' and h' are independent.) Since f' and h' are complementary, $h'(a, b\Delta 1) \neq h'(\emptyset, b\Delta 1)$. Since coordinatization preserves monotonicity, $h'(\emptyset, b\Delta 1) \subset h'(a, b\Delta 1)$. Now we have $h'(\emptyset, b\Delta 1) \subset h'(a, b\Delta 1) = h'(\emptyset, b\Delta 0) \subset h'(a, b\Delta 0)$. We can choose $b\Delta i + 1$ such that $h'(a, b\Delta i + 1) = h'(\emptyset, b\Delta i)$. This defines an infinite sequence of sets, each of which is a proper subset of the previous one. This is not possible when the domain is finite sets. Thus, we have arrived at a contradiction. ∎

3.5 Conclusion

We have considered the relationship between independent and complementary mappings. We have studied the issue on databases on finite domains with monotonic mappings. We have shown that given one mapping, there is at most one other mapping that is independent and complementary to it. The domains and ranges are all finite power sets. An intersection mapping intersects each domain element with a fixed set (called the intersection set) to produce the result. Given a pair of mappings that are monotonic, independent, and complementary, the two mappings are equivalent to intersection mappings where the intersection sets are a partition of the generator of the power set. Because our views are monotonic mappings, these results show some shortcomings of the concepts of complementary views.

Chapter 4

Update Semantics

A user accessing a database through a view should not be concerned that operations expressed against the view must be translated into operations expressed against the database. Rather, the user should be able to treat the view the way a user would treat a stored database relation. The view is essentially the complete representation of the user's world. Queries are well understood for both databases and views, so we need not consider them further. We use the same three operators to update through a view as we use for updating an ordinary database: insert, delete, and replace. The semantics of these operators on a view is defined based on the semantics of these operators on an ordinary database.

4.1 Direct Relation Updates

In order to understand what an update expressed against a view should mean, we need to understand what an update expressed directly against the database means. We use the same three operators for view updates as for database updates: insert, delete, and replace. An insertion supplies a completely specified tuple to be added to a relation. The insertion can proceed only if it does not violate any constraints on the database. In particular, if there is a functional dependency defined, there must not already be a tuple whose key matches the key of the tuple to be inserted. We specifically do not permit insertion of a tuple that matches exactly an existing tuple in that relation; we say that the update is rejected rather than it having no effect. The result of the insertion is the union of the original relation and the tuple to be inserted.

A deletion supplies a completely specified tuple to be removed from a relation. That tuple must exist in the relation to be removed. The result of the deletion is the set difference of the original relation minus the tuple to be removed. As a shorthand, a deletion may specify a condition for testing completely specified tuples to be deleted. The effect is as if each of these tuples were deleted individually.

A replacement supplies an existing completely specified tuple and its completely specified replacement. The replaced tuple must already exist and the replacement tuple must not already exist. If the tuple to be replaced has the same key as the replacement tuple, the replacement cannot violate a functional dependency in that relation. It may, however, violate other constraints, if there are any. If the tuple to be replaced has a different key from the replacement tuple, there must not already be a tuple in that relation with a key matching the replacement tuple, as that would violate the functional dependency. A request to replace a tuple with itself is not specifically rejected; rather it is a valid operation that has no effect. The result of the replacement is the removal of the tuple to be replaced and the addition of the replacement tuple. Note that a deletion followed by an insertion is not the same as a replacement: The former requires an intermediate consistent state that the latter does not.

4.2 View Updates

We use the same three operations for view updates as for database updates: insert, delete, and replace. The user treats the view as if it were an ordinary relation, although it is not necessarily in Boyce-Codd Normal Form. Consequently, when the user requests an insertion, deletion, or replacement, the view should change correspondingly.

In a projection view where the key of the relation does not appear in the view, the question of handling duplicates arises. Two database tuples may be mapped by the view into the same view tuple. If duplicates appear in the view (i.e., duplicates are not removed during the projection), then how does a change to one view tuple affect the other matching view tuples? Since the view tuples in question are identical, if one view tuple satisfies the selection clause for an update, they all will. Consequently, the request to delete one of a set of duplicate view tuples will result in deletion of the entire set of view tuples with the attendant changes occurring to the corresponding underlying database tuples. If duplicates do not appear in the view (i.e., duplicates are removed during the projection), then how does a change to the view tuple affect the corresponding underlying (database) tuples? If a view tuple is to be deleted, that view tuple must no longer appear in the view. This may require multiple changes to the database in order to accomplish the removal of the view tuple, corresponding to the duplicate view tuples that were collapsed into one view tuple. Since a change to a view tuple affects all identical view tuples equally, the effect on the database will not depend on whether duplicates are removed from the view. Note that most of the time we will not allow database keys to be removed from the view by projections, so this problem will not ordinarily arise.[1]

4.2.1 Translations vs. translators

A view update translation is a sequence of database updates that implement a view update request. There may be several view update translations for a given view update request.

A view update translator is a mapping from view update requests into translations of these view update requests. A translator would take the user's view update requests and translate them into database update requests, which could then be processed by the database system. Thus, a view update translation facility would be a useful adjunct to a database system which has a view definition facility.

We shall provide a list of view update translators grouped into broad classes. We do not claim that these are the only view update translators possible. Rather, we show that, for each view update request, all possible translations of that view update request into sequences of database update requests that satisfy our 5 criteria are obtained from these view update translators.

We can obtain other view update translators from the translators we define. One translator can be obtained from another by rejecting some updates and translating the others. Given two translators, there are translators that use some translations from translator and some from the other. Using these two translator transformations,

[1]We do cover projections in which complete keys do not appear in Sections 5.2.3 through 5.2.6.

from the view update translations we define, we can generate the complete set of view update translators whose translations that satisfy our 5 criteria.

In order that the facility for handling view updates may choose among these alternatives at view update time, we propose that the database adminstrator (DBA) provide additional semantics during view definition time. These additional semantics permit the view update facility to choose one of these translators.

4.3 Faithfulness criteria

A translation of a particular view update request is characterized by three sets consisting of the insertions, deletions, and replacements applied to the underlying database. The insertions and deletions are each described by a set containing the affected tuples. The replacements are described by a set of ordered pairs of old and new tuples. These sets all contain the exact tuples, specifying all attributes. We will consider two translations equivalent if they have the same effect on the database. In practice, the equivalence can result from converting a pair of an insertion and a deletion into a replacement, or from swapping the replacement tuples from a pair of replace operations. Formally, let the set of removed tuples be the union of the set of deleted tuples and the set of replaced tuples; similarly, the set of added tuples is the union of the set of inserted tuples and the set of replacement tuples. Then two translations are equivalent if their respective added and removed sets are equal. Recall that the translations are valid when the implement the request exactly (without additional changes to the view).

All of the candidate update translations are to satisfy the following 5 criteria in addition to being valid.

1. "No database side effects." The only database tuples affected have keys that match their respective values in the tuples mentioned in the view update request. (This is part of the rectangle rule [Dayal 78, 79, 82]). Note that this requires the key of each relation affected to appear in the view. In particular, this means that if the key of a tuple changes, the old and new keys must appear in the respective positions of the view update request.

2. "Only 'one step' changes." Each database tuple is affected by at most one step of the translation for any single view update request. Specifically, a translation

cannot replace an inserted tuple, or delete a replaced tuple, or replace a tuple twice in succession. This rule implies that there is no ordering imposed on the individual database updates that translate a view update.

3. "Minimal change: no unnecessary changes." There is no valid translation that implements the request by performing only a proper subset of the database requests. (Note that we are concerned with the set of database operations; a deletion is not simpler than a replacement that replaces the same tuple since the replacement is a single request.)

4. "Minimal change: replacements cannot be simplified." Consider two alternative database replacement requests where both specify replacing the same tuple. A database replacement that does not involve changing the key is simpler than one where the key changes. A database replacement that changes a proper subset of the attributes changed by another database replacement is simpler.

5. "Minimal change: no delete-insert pairs." We do not allow candidate translations to include both deletions and insertions on any one relation, as they may be converted into replacements, which we consider simpler. Thus candidate translations may contain either deletions or insertions for any relation, but not both, in addition to replacements. A translation may, however, contain an insertion into one relation and a deletion from another relation.

Let us consider the implications of these five criteria. Criterion 1 (no database side effects) requires that any change to the database affect only tuples that are relevant to the view update. This requirement is not as stringent as maintaining a constant complement. Both requirements are intended to eliminate unintended effects on other users. The constant complement method takes a fixed notion of some other user and prevents any actions through our view from affecting the other user, which is contrary to the concept of a shared database and precludes some reasonable updates or update translators.

Criterion 2 (only 'one step' changes) eliminates two types of anomalies. We do not want a tuple to be replaced by two separate tuples; it would have disappeared after the first of these replacements. We also do not want a tuple to undergo multiple separate changes, as these could be combined into a single change. For example, we do not want a tuple to be replaced only to be deleted in its new form. Rather, the original tuple should be deleted.

Criteria 3, 4, and 5 require that the updates be minimal. This takes three forms: we do not do any unnecessary operations, the operations we do perform are the simplest ones possible, and we never do in two steps what can be done in one. The only operation we can simplify is a replacement operation, and it is simplified by not changing the key, or by changing fewer attributes. We consider a one-step replacement operation simpler than a two-step deletion-insertion pair. This allows us to get at the essence of the necessary changes.

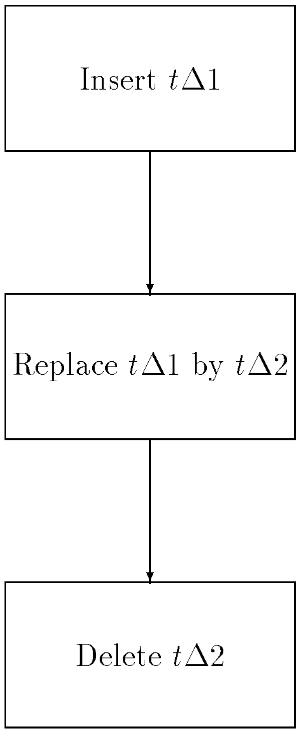
The purpose of our five criteria is to permit all possible changes, but only in their simplest forms. If changing a particular attribute value is sufficient, we want to consider that in preference to changing that attribute in addition to others. It is certainly possible to translate a view update request by performing additional changes to the database, but there are endless possibilities for such elaboration. If we are to achieve our goal of characterizing the possibilities, we must restrict ourselves to the simplest ones, which capture the essence of the changes in the more elaborate translations.

Based on the definitions of added and removed sets above, one translation is at least as simple as another if its added and removed sets are subsets of those of the other translation.

Theorem 4.1 *For every given view update request and for every valid translation, there is (at least one) translation at least as simple that satisfies the 5 criteria.*

Proof: Let us define a lattice of view update translations that implement the desired view update. Criteria 3 and 4 clearly define part of the ordering on the lattice.

For each translation that does not satisfy Criterion 2, there must be a pair of database operations affecting the same database tuple. We will create a directed graph where each operation is a node and there are edges between operations that affect a database tuple created by a previous operation. We want to transform the graph to remove all edges. Edges emanate from insert and replace operations and terminate at delete and replace operations. Since the translation is described as a sequence of operations, no node can have out-degree greater than one, nor may the graph have a cycle. If either of these events occur, the translation has no ordering of the operations that represents a valid sequence for the database.



Graph of Operations on Tuples

We consider each of the leaf nodes. For leaf nodes that are delete operations, we repeatedly delete the node we are at and then visit its parent node until we reach a root node. For leaf nodes that are replace operations, we repeatedly collapse the node we are at with its parent node until we reach a root node. We collapse the operation "replace $t\Delta2$ by $t\Delta3$" preceded by "replace $t\Delta1$ by $t\Delta2$" by substituting for these and the edge separating them the operation "replace $t\Delta1$ by $t\Delta3$." Similarly, we collapse the operation "replace $t\Delta2$ by $t\Delta3$" preceded by "insert $t\Delta2$" by substituting for these and the edge separating them the operation "insert $t\Delta3$." The resulting graph has no edges and satisfies Criterion 2; it still performs the desired view update because it ends up with the same database state as the original translation when it starts with the same database state.

For each translation that does not satisfy Criterion 1, we change all replacements that violate Criterion 1 to their respective delete-insert pairs. (This will temporarily violate Criterion 5.) For the other cases, we remove all steps of the translation that do not involve tuples with keys that match their respective values (same attribute names) in the view update request. The resulting translation clearly satisfies Criterion 1, but we must show that it still performs the desired change to the view. Since select, project, and join views are monotonic, the only database tuples that affect the view tuples to be changed are the corresponding underlying tuples. But those are the ones with keys matching the values in the view update request.

For each translation that includes a deletion and an insertion operation in the same relation, we replace all such pairs by a replacement of the deleted tuple by the inserted tuple. Such a translation now satisfies Criterion 5 and still implements the desired view update.

We observe that the lattice define is sound (the partial order is well defined). Since there are only a finite number of operations and attributes, the lattice must be finite. Therefore, there must be one or more "least" elements. Such a least element satisfies all five criteria. ∎

Theorem 4.2 *The five criteria are independent.*

Proof: We shall give examples of views where all but one criterion are satisfied.

Criterion 1. The view selects tuples on relation AB with odd numbers for component `a`n in domain A where AßB. View update is insert ⟨`a1, b1`⟩. Database update consists of replace ⟨`a2, b1`⟩ by ⟨`a1, b1`⟩. This affects ⟨`a2, b1`⟩, which violates Criterion 1 but not the others.

Criterion 2. View is identity view on relation AB where AßB. View update is replace ⟨`a1, b1`⟩ by ⟨`a1, b3`⟩. Database update consists of replace ⟨`a1, b1`⟩ by ⟨`a1, b2`⟩ and replace ⟨`a1, b2`⟩ by ⟨`a1, b3`⟩. This satisfies all criteria but 2.

Criterion 3. View is join of relation AB with relation BC, where AßB and BßC. Deletion of view (ABC) tuple ⟨`a1, b1, c1`⟩ can be accomplished by deletion of ⟨`a1, b1`⟩ alone. Adding the deletion of ⟨`b1, c1`⟩ results in a translation that satisfies all criteria but 3.

Criterion 4. View is projection of relation ABC where AßBC and only attributes A and B appear in the view. Replacement of view tuple ⟨`a1, b1`⟩ by ⟨`a1, b2`⟩ can be done by replacing database tuple ⟨`a1, b1, c1`⟩ by ⟨`a1, b2, c1`⟩. Replacing database tuple ⟨`a1, b1, c1`⟩ by ⟨`a1, b2, c2`⟩ satisfies all criteria but 4.

Criterion 5. View is projection of relation ABC where AßBC and only attributes A and B appear in the view. Replacement of view tuple ⟨`a1, b1`⟩ by ⟨`a1, b2`⟩ can be done by replacing database tuple ⟨`a1`, `b1`, `c1`⟩ by ⟨`a1`, `b2`, `c1`⟩. Deleting database tuple ⟨`a1`, `b1`, `c1`⟩ and inserting database tuple ⟨`a1`, `b2`, `c2`⟩ satisfies all criteria but 5.

Thus, each criterion can be violated without violating the others. ∎

4.4 Exactness of Translation

One measure of a view update translation is whether the view update request is performed exactly (i.e., with no view side effects). In the diagram, the question is whether the diagram commutes; is $U(V(DB)) = V(DB')$?

$$\begin{tikzcd} V(DB) \arrow[r, "U"] \arrow[dr, phantom, "\Downarrow T"] & U(V(DB)) \overset{?}{=} V(DB') \\ \mathrm{DB} \arrow[u, "V"'] \arrow[r, "T(U)"] & T(U)(DB) = DB' \arrow[u, "V"'] \end{tikzcd}$$

We shall not allow view side effects in select and project views, as it is possible to translate view updates for these views without side effects and still satisfy our 5 criteria. However, we are forced to allow side effects in join views for some updates. For example, consider the following relations AB and BC and view ABC (the natural join of AB and BC), where AßB and BßC.

AB		*BC*		*ABC*		
A	B	B	C	A	B	C
a_1	b_1	b_1	c_1	a_1	b_1	c_1
a_2	b_1			a_2	b_1	c_1

If we request that view tuple $\langle a_1, b_1, c_1\rangle$ be replaced by $\langle a_1, b_1, c_3\rangle$, the translation must include replacing $\langle b_1, c_1\rangle$ by $\langle b_1, c_3\rangle$ as a consequence of the view definition. This necessarily has the side effect of replacing the view tuple $\langle a_2, b_1, c_1\rangle$ by $\langle a_2, b_1, c_3\rangle$. The actual effect on the view is the result of decomposing the user's request into a series of requests, each confined to the attributes of a single underlying BCNF relation.

The views side effects permitted in join views are always to maintain functional dependencies. These functional dependencies in the database are visible in the view and must be maintained there as well. When a view update request (replacement as well as insertion) is presented that would violate a functional dependency, the conflicting tuples must be changed in accordance with the functional dependencies if the update is to be accepted.

Chapter 5

Single Relation Updates

We will consider updates to views consisting of selections and projections of a single relation in Boyce-Codd Normal Form (BCNF). In particular, note that these views do not have joins. We will consider joins in the next chapter.

5.1 Selections

Let us consider the relation `EMP` which contains each employee's number, name, location, and whether the employee is a member of the company baseball team. The company has two locations: New York and San Francisco. Baseball team members must be employees.

The personnel manager, Susan, in New York has the following view definition:

View P:
```
   Select *
   From EMP
   Where Location="New York"
```

She requests the deletion of employee #17 from her view. A reasonable translation of this request is to delete the employee record from the underlying database. Thus, we have translated a view deletion into a database deletion. If the employee was a member of the baseball team, he has been removed from that also.

The baseball team manager, Frank, has the following view definition:

View B:

```
Select *
From EMP
Where Baseball="Yes"
```

He requests deletion of employee #14 from his view. It is unreasonable to delete the employee tuple from the underlying database (unless you believe that baseball is all-important). A reasonable translation of this view deletion request is to replace the Baseball attribute of the underlying database tuple with a "No." Thus, we have translated a view deletion into a database replacement.

One might argue that the Frank's view deletion request should have been a replacement. However, this would mean requesting the replacement of a tuple in the view with a view tuple that did not appear in the view. Then Frank's request would not be valid in the view, as the replacement tuple could not possibly be a view tuple. In addition, Frank would have to make a distinction between deletion and replacement that he could not discern by looking at the effects through his view.

It is possible to translate the Susan's request by moving employee #17 to California. We doubt that the California manager would be pleased by such an implementation. Rather, such a request should be issued by someone authorized to access the entire relation (as a replacement request): someone who can see the effects of that request.

We see that a view deletion request is sometimes best translated into a database deletion request and at other times into a database replacement request. As we shall see, similar alternatives arise for insertion and replacement. We suggest that additional semantics be used to choose among the various alternatives, and this will be discussed in Chapter 7.

5.1.1 Class of Selection Views

We will first consider views consisting of selections of a single Boyce-Codd Normal Form relation. We will then consider the projection views and selection-and-projection views.

We will deal with select and project views on a single relation with a single key dependency. Let R be the set of attributes in the relation $\mathbf{R}$, and let K be the set of attributes in the key. We will assume that the functional dependency KßR is the *only* consistency constraint on **R**. Observe that since the relation may only have a key constraint, the database has already undergone normalization.

Let us first consider the selection condition. The selection condition is a conjunction of terms of the form $A \in s$ (or equivalently, $A \notin e$), where s and e are sets of constants in the domain of A. (Note that s´e is equal to the domain of A.) We call the values in s *selecting values*, and the values in e *excluding values*. For non-selecting attributes the set of selecting values is the entire domain and the set of excluding values is the empty set. We call the attributes appearing in the selection condition *selecting attributes*. If the selection condition is "true" (i.e., an empty conjunction), the set of selecting attributes is empty, the sets of selecting values are the entire domains, and the sets of excluding values are empty. This type of selection condition allows attributes to be treated independently in view updates.

We will define a class of view update translators with certain desirable properties. We then enumerate the members of this class. We will also give some completeness results.

We handle only single view tuple update requests of the following forms: (1) Insert a single, fully defined tuple. (2) Delete a single tuple by supplying its key. (3) Replace a key by supplying the values of all attributes of the original and replacement view tuples.

The requested view updates must be valid as specified on the view, otherwise they are rejected before ever being considered by the view update translator. That is, the view update request would be valid if the view were an ordinary relation, with only the constraints visible in the view required to hold (in this case, the FD must hold if the key appears in the view). Note that we use the term "update" to mean insertion, deletion, or replacement, while we use "replace" to refer to the operation of removing a tuple from the view (or database) and substituting another for it.

While the only constraint we will consider is a single key dependency, we do allow additional constraints to be imposed on the view or on the database. We attempt to translate the request ignoring the additional constraints and then test whether the translation satisfies those constraints. If not, we reject the request and make no change to the database. Inter-relation constraints, such as inclusion dependencies, are examples of constraints that are amenable to such treatment and can easily be tested [Keller 81].

5.1.2 Translation of Update Requests

We will consider the general case of translating update requests on a selection view into updates to the underlying database. We will deal with single tuple insertions first. We will follow that with single tuple deletions. Finally, we will deal with single tuple replacements.

With selection views, we can compare expressing the update against the view with expressing the update against the underlying database. Any single tuple request that is valid against the database is also valid against the view if all tuples affected satisfy the selection criteria. Performing the same request directly against the database is one of the possible translations of the request when expressed against the view. However, there can be other translations of such view updates. Furthermore, there are valid view update requests (for which translations to database updates exist) that are not valid when expressed against the database. For example, consider the view update request to insert a view tuple that has the same key as a database tuple that does not appear in the view. This view update request is valid and can be translated by replacing that database tuple which does not appear in the view with the view tuple to be inserted; inserting the view tuple directly into the database is not a valid database request. However, the situation is not this simple when considering more complex views.

5.1.3 Translations of Insertion Requests

The request is to insert a single fully specified view tuple. The new view tuple must be a valid view tuple—satisfy the selection condition—and not conflict with any existing view tuple. We would like algorithms that define translations of view updates into database updates that satisfy the five criteria for acceptable translations that we specified earlier. Algorithm S-I-1 handles the case where the new view tuple does not have the same key as an existing database tuple, while algorithm S-I-2 handles the case where the new view tuple matches the key of a database tuple.

Algorithm S-I-1: If the new view tuple does not conflict with (i.e., does not have the same key as) any existing *database* tuple, then insert the desired tuple into the database else reject the update request.

Lemma 5.1 *The updates generated by Algorithm S-I-1 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuple affected is the inserted view tuple. (2) There is only one database request. (3) The only database request is an insertion, so clearly doing less could not implement the view update request. (4) There are no database replacement requests. (5) There are no database deletion requests. ∎

We observe that this update preserves the following complement:

```
Select *
From R
Where NOT (<condition>)
```

Algorithm S-I-2: If there is a database tuple whose key matches that of the new view tuple, then replace that database tuple with the new view tuple (in the database) else reject the update request.

Note that the database tuple with the matching key did not appear in the view. This requires that the selection condition must depend in part on attributes that are not part of the key.

Lemma 5.2 *The updates generated by Algorithm S-I-2 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the inserted view tuple and the replaced database tuple, both with the same key value. (2) There is only one database request. (3) The only database request is a single replacement, so clearly doing less could not implement the view update request. (4) There is a single database replacement request, and it changes only those attributes that differ between the old database tuple and the new view tuple. No replacement could be simpler because there are no other candidate tuples for replacement with the same key. (5) There are no database deletion requests. ∎

In the case where the replacement changes a single attribute value (for attribute A) which is mentioned in a selection term that excludes a single value, such as `Baseball`$\neq$`"No"`, (possibly in conjunction with other selection terms), this update translator generates one translation that preserves the following complement:

```
Select *-A
From R
Where True
```

For some views, such as those with a selecting attribute with more than one excluded value, there is no complement that this translator will always hold constant. If it is desired to maintain a constant complement, some view update requests that could be translated must be rejected.

Theorem 5.1 *The set of update translations that satisfy the five criteria for candidate update translations for individual view insertions are precisely those that are generated by Algorithms S-I-1 and S-I-2.*

Proof: The two previous lemmata show that the translations generated by the two algorithms satisfy the five criteria.

We will now show that there are no other translations satisfying the five criteria. We first observe that every view update valid in the view can be translated by either Algorithm S-I-1 or by Algorithm S-I-2. Because the view is monotonic—a view is monotonic if view formation preserves inclusion (i.e., $A \subseteq B ß v(A) \subseteq v(B)$)—a view insertion cannot be translated solely by database deletions. Therefore, any set of database updates implementing the view insertion request must contain at least one insertion or replacement. Precisely one of these requests must refer to the new view tuple (by Criterion 2). If it is a replacement, the replacement tuple must be the new view tuple as no subsequent database request can affect that tuple in this view update request. That request is generated by Algorithm S-I-2, as the replaced tuple must have a matching key (by Criterion 1). If it is an insertion, the tuple inserted must be the new view tuple as no subsequent database request can affect that tuple in this view update request. That request is generated by Algorithm S-I-1. However, by Criterion 5, the translation may not separately request deletion of a conflicting database tuple—the one replaced in Algorithm S-I-2. Such a sequence would be simplified to the acceptable translation from Algorithm S-I-2. ∎

Recall that a view update translator is a mapping from view update requests into translations of these view update requests. A translator would take the user's view update requests and translate them into database update requests, which could then be processed by the database system. Algorithms S-I-1 and S-I-2 define view update translators that handle mutually exclusive sets of view updates. We could define a new view update translator that translates each view update by the algorithm of the two that can handle it. This combination translator has the translations of the two original algorithms. Such a combination algorithm, unlike Algorithms S-I-1 and S-I-2, does not have a counterpart in the deletion case, as we will see.

5.1.4 Translations of Deletion Requests

The request is to delete a single fully specified view tuple. The deleted view tuple must be a valid view tuple. Algorithm S-D-1 deletes the underlying database tuple while algorithm S-D-2 replaces it. Algorithms S-D-1 and S-D-2 are the analogs of algorithms S-I-1 and S-I-2, respectively.

Algorithm S-D-1: Delete the underlying database tuple.

Lemma 5.3 *The updates generated by Algorithm S-D-1 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuple affected is the deleted view tuple. (2) There is only one database request. (3) The only database request is an deletion, so clearly doing less could not implement the view update request. (4) There are no database replacement requests. (5) There are no database insertion requests. ∎

We observe that this update also preserves the following complement:

```
Select *
From R
Where NOT (<condition>)
```

Algorithm S-D-2: Replace the value of one non-key attribute mentioned in the selection condition with a value that does not satisfy (is excluded by) the selection condition. If the only attributes appearing in the selection condition are part of the key, this algorithm is not applicable (it would violate Criterion 1).

Note that the replacement database tuple does not appear in the view.

Lemma 5.4 *The updates generated by Algorithm S-D-2 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the deleted view tuple and the replacement database tuple (whose key matches that of the deleted view tuple). (2) There is only one database request. (3) The only database request is a single replacement, so clearly doing less could not implement the view update request. (4) The key is not changed. There is a single database replacement request, and it changes only those attributes that differ between the old database tuple and the new view tuple. No replacement could be simpler because there is only one (non-key) attribute is changed. (5) There are no database insertion requests. ∎

The form of the selection condition (conjunction of attribute selecting conditions) implies that it is sufficient to change only one attribute value to cause the tuple to disappear from the view. Translation Criterion 3 implies that we do not change more than one attribute value if changing one value is sufficient and the key does not change. In the case where the replacement changes a single attribute value (for attribute A) which is mentioned in a selection term that excludes a single value, such as `Baseball≠"No"`, (possibly in conjunction with other selection terms), this update translator preserves the following complement:

```
Select *-A
From R
Where True
```

For some other views, Algorithm S-D-2 does not preserve any complement. If it is desired to preserve a complement, some of the view updates that would otherwise be accepted must be rejected.

There is an important difference between Algorithms S-I-2 and S-D-2. When translating an insertion into a replacement, the new tuple is known. When translating a deletion into a replacement, it is the old tuple that is known; thus, which attribute should be changed and what value it should be given may not be constrained by the view update request and the selection condition.

Theorem 5.2 *The set of update translations that satisfy the five criteria for candidate update translations for individual view deletions are precisely those that are generated by Algorithms S-D-1 and S-D-2.*

Proof: The two previous lemmata show that the translations generated by the two algorithms satisfy the five criteria.

We will now show that there are no other translations satisfying the five criteria. We first observe that every view update valid in the view can be translated by Algorithm S-D-1 and sometimes by Algorithm S-D-2. Because the view is monotonic, a view deletion cannot be translated solely by database insertions. Therefore, any set of database updates implementing the view deletion request must contain at least one deletion or replacement. Precisely one of these requests must refer to the deleted view tuple. If it is a replacement, the replaced tuple must be the deleted view tuple, as no preceding database request can affect that tuple in this view update request by Criterion 2. That request is generated by Algorithm S-D-2, and the replacement tuple has a matching key if possible (by Criteria 1 and 4). If it is a deletion, the tuple deleted must be the deleted view tuple as no preceding database request can affect that tuple in this view update request. That request is generated by Algorithm S-D-1. ∎

Algorithms S-I-1 and S-I-2 can be applied at mutually exclusive database states. That is, a database state has at least one valid translation generated by Algorithm S-I-1 or S-I-2 *but not both*. Thus, we could define an Algorithm S-I-1´2 that includes the translations of both Algorithms S-I-1 and S-I-2. Unlike Algorithms S-I-1 and S-I-2, Algorithms S-D-1 and S-D-2 are alternatives sometimes applicable to the same database state. Consequently, there is no algorithm for deletion that can be constructed along the lines of Algorithm S-I-1´2. Note that Algorithm S-D-1 can *always* be applied while Algorithm S-D-2 can only *sometimes* be applied.

5.1.5 Translations of Replacement Requests

The request is to replace one single fully specified view tuple with another. Both view tuples must satisfy the selection condition. The new view tuple must not conflict with any view tuple other than the one it replaces. There are five algorithms. Algorithm S-R-1 handles the case where the key does not change. The other four algorithms handle the case where the key does change. There are two options for how the old tuple gets removed; these correspond to the two deletion algorithms. There are also two options for how the new tuple gets added; these correspond to the two insertion algorithms. All four combinations of these two options are possible.

Algorithm S-R-1: If the key does not change, then perform the replacement on the database else reject the update request

Note that if the key does not change, there is no possibility of a conflict with any tuple not appearing in the view.

Lemma 5.5 *The updates generated by Algorithm S-R-1 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the replaced and replacement view tuple. (2) There is only one database request. (3) The only database request is a single replacement, so clearly doing less could not implement the view update request. (4) There is a single database replacement request, and it changes only those attributes that differ between the old view tuple and the new view tuple. No replacement could be simpler because there are no other candidate tuples for replacement with the same key. (5) There are neither database insertion nor database deletion requests. ∎

We observe that this update preserves the following complement:

```
Select *
From R
Where NOT (<condition>)
```

Algorithm S-R-2: If the key changes, but there is no database tuple whose key matches that of the new view tuple, then perform the specified replacement on the database directly. Otherwise, reject the request.

Algorithm S-R-2 will not allow changes to database tuples not appearing in the view.

Lemma 5.6 *The updates generated by Algorithm S-R-2 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the replaced view tuple and the replacement view tuple. (2) There is only one database request. (3) The only database request is a single replacement, so clearly doing less could not implement the view update request. (4) There is a single database replacement request, and it changes only those attributes that differ between the old database tuple and the new view tuple. No replacement could be simpler because there are no candidate tuples for replacement with the same key. (It is possible to use a different replacement tuple with the same key, but that would require an additional database request, as we will see in Algorithm S-R-4.) (5) There are neither database insertion nor database deletion requests. ∎

Algorithm S-R-3: If the key changes and there is a database tuple whose key matches that of the new view tuple, then replace that database tuple with the new view tuple (in the database) and delete the replaced view tuple from the database. Otherwise, reject the update request.

Algorithm S-R-3 changes a database tuple that does not appear in the view. If we consider the view replacement as a view deletion and a view insertion, then the dichotomy between Algorithms S-I-1 and S-I-2 parallels that between Algorithms S-R-2 and S-R-3 (respectively).

Lemma 5.7 *The updates generated by Algorithm S-R-3 satisfy the five criteria for candidate update translations.*

Proof: (1) The database update is the deletion of the original view tuple and the replacement of a database tuple not appearing in the view with the new view tuple. (2) The three changed database tuples are all distinct. (3) All three tuples must be involved in the database update for the update to be translated. This is because the replaced view tuple must not appear in the view (so it must not appear in the database), the replacement view tuple must appear in the view (so it must now appear in the database), and the conflicting database tuple cannot be in the database along with the replacement view tuple (so something must be done to it also). At least two requests are necessary to affect three tuples, and there are exactly two requests. (4) The single database replacement request does not change the key. It can only be simplified by changing fewer attributes. However, there are no alternative replaced or replacement tuples with the same key. Therefore, the replacement cannot be simplified. (5) There are no database insertion requests. ∎

Algorithms S-R-2 and R-3 assume that no tuple remains in the database with a key matching that of the replaced view tuple. However, it is possible to replace (in the database) the replaced view tuple with a tuple with a matching key that does not satisfy the selection criteria. This parallels the dichotomy of Algorithms S-D-1 and S-D-2 for deletion. We obtain two algorithms using the same conditions of Algorithms S-R-2 and S-R-3, respectively.

Algorithm S-R-4: If the key changes, but there is no database tuple whose key matches that of the new view tuple, then replace (in the database) the replaced view tuple by changing one non-key attribute mentioned in the selection condition with a value that does not satisfy (is excluded by) the selection condition. Note that this algorithm is not applicable (it would violate Criterion 1) when the only attributes appearing in the selection condition are part of the key. Also, insert the replacement view tuple into the database. Otherwise, reject the update request.

Lemma 5.8 *The updates generated by Algorithm S-R-4 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the replaced view tuple, a new database tuple with a matching key, and the replacement view tuple. (2) The replacement view tuple (the inserted database tuple) does not match the two tuples mentioned in the database replacement. (3) Neither request is sufficient by itself. (4) The only database replacement request changes only a single non-key attribute. (5) There are no database deletion requests. ∎

Algorithm S-R-5: If the key changes and there is a database tuple whose key matches that of the new view tuple, then replace that database tuple with the new view tuple (in the database). Also, replace (in the database) the replaced view tuple by changing one non-key attribute mentioned in the selection condition with a value that does not satisfy (is excluded by) the selection condition. Note that this algorithm is not applicable (it would violate Criterion 1) when the only attributes appearing in the selection condition are part of the key. Otherwise, reject the update request.

Lemma 5.9 *The updates generated by Algorithm S-R-5 satisfy the five criteria for candidate update translations.*

Proof: (1) There four tuples affected in the database update. The original view tuple is replaced by a database tuple with a matching key. The replacement view tuple replaces a database tuple with a matching key. (2) The four database tuples are clearly distinct. (3) Each of the replacements is needed: one removes the replaced view tuple, the other inserts the replacement view tuple. (4) Both database replacements do not change the key. The database replacement that inserts the new view tuple can only be

Delete / Insert	Delete replaced tuple (cf. algorithm S-D-1)	Replace (in database) replaced view tuple (cf. algorithm S-D-2)
No conflict for replacement view tuple (cf. algorithm S-I-1)	Algorithm class S-R-2 One replacement	Algorithm class S-R-3 Replace old view tuple Insert new view tuple
There is a database tuple whose key matches the replacement view tuple (cf. algorithm class S-I-2)	Algorithm class S-R-4 Delete old view tuple Replace new view tuple	Algorithm class S-R-5 Replace both old and new view tuples

How algorithms for replacement when the key changes are related to deletion and insertion algorithms

simplified by changing fewer attributes. However, there are no alternative replaced or replacement tuples with the same key. Therefore, that replacement cannot be simplified. The other database replacement request changes only a single non-key attribute. (5) There are neither database deletions nor database insertions. ∎

Algorithm S-R-1 is the only one possible when the replacement does not change the key. When the view replacement changes the key, we have two options for handling the replaced view tuple corresponding to Algorithms S-D-1 and S-D-2, and two situations involving the replacement view tuple corresponding to Algorithms S-I-1 and S-I-2. The four replacement algorithms for use when the key changes are described by the table.

Theorem 5.3 *The set of update translations that satisfy the five criteria for candidate update translations for individual view replacements are precisely those that are generated by Algorithms S-R-1, S-R-2, S-R-3, S-R-4, and S-R-5.*

Proof: The five previous lemmata show that the translations generated by the five algorithms satisfy the five criteria.

We will now show that there are no other translations satisfying the five criteria. We consider the three exclusive cases based on the enabling conditions for Algorithm S-R-1, for Algorithms S-R-2 and S-R-4, and for Algorithms S-R-3 and S-R-5.

Case 1. The key does not change. (Algorithm S-R-1.)

The view update requests replacement of one tuple by another. The replaced tuple must have been in the database and the replacement tuple must now be in the database. Since the two tuples have the same key, the replaced tuple must no longer be in the database. We have already seen that the only database operations that remove a tuple from the database are delete and replace.

If the operation is replace (the tuple to be removed by something else), the request where the replacement tuple is the new view tuple is strictly simpler than all other possible replacements. In that case, the database replacement must be precisely the requested view replacement. No other operations are possible (in addition) as the database replacement is sufficient. This is the translation generated by Algorithm S-R-1.

If the operation is delete, we consider how the new view tuple appeared in the database. A new tuple can only appear in a database by insertion or replacement. If it was by insertion, this insertion and (previous) deletion are together sufficient, and the two are equivalent to (and not as simple as, by Criterion 5) the replacement generated by Algorithm S-R-1. On the other hand, if the new view tuple got into the database by replacement, an argument similar to that in the previous paragraph will show that the replaced database tuple must have been the replaced view tuple. With the precondition (of this paragraph) that this tuple was deleted (rather than replaced), we have a contradiction by Criterion 2.

Case 2. The key changes, but there is no (existing) database tuple whose key matches that of the new view tuple. (Algorithms S-R-2 and S-R-4.)

Let us consider what happens to the replaced view in the database. It can either be deleted or replaced. If it is deleted, we consider how the replacement view tuple got in the database. It could have been inserted or it could be a replacement. If that tuple was inserted, then the deletion and insertion are equivalent to (not as simple as) the replacement generated by Algorithm S-R-2. If it is a replacement, it must have replaced a database tuple with a differing key. By Criterion 1, the database tuple replaced must be the view tuple replaced. With the assumption that this tuple was deleted (rather than replaced), we have a contradiction by Criterion 2.

Alternatively, the replaced view tuple is replaced in the database. The replacement database tuple can have a matching or non-matching key. If a non-matching key, by Criterion 1, the replacement database tuple must be the replacement view tuple resulting in Algorithm S-R-2. If the database replacement does not change the key, the simplest replacements change only one attribute, as in Algorithm S-R-4. In this case, we must also consider how the replacement view tuple got in the database. Again, it could have been inserted or it could be a replacement. If that tuple was inserted, then the replacement and insert is as generated by Algorithm S-R-4. If it is a replacement, it must have replaced a database tuple with a differing key. By Criterion 1, the database tuple replaced must be the view tuple replaced. However, we assumed that the original view tuple was replaced by a tuple with a matching key. Consequently, we have a contradiction by Criterion 2.

Case 3. The key changes, and there was a database tuple (that did not appear in the view) whose key matches that of the new view tuple. (Algorithms S-R-3 and S-R-5.)

Criterion 1 states that affected tuples must have keys that match the tuples mentioned in the view update. The two tuples with the key of the replacement view tuple are the matching database tuple that did not appear in the view and the view tuple itself. There is the possibility that there is a new database tuple (not appearing in the view) whose key matches that of the replaced view tuple. Thus, we have exactly three or four tuples affected by the translation of the view update. If only three tuples are affected, the combination of insertions, deletions, and replacements is equivalent to (and not as simple as) the translations generated by Algorithm S-R-3. If four tuples are affected, this extra tuple must be a replacement rather than an insertion (which would be an extraneous operation). It must be a replacement of the replaced view tuple and only one attribute may change as a consequent of Criterion 4. ∎

5.2 Projections

We will deal with projection views on a single relation with a single key dependency. Let A be the set of attributes in the relation R, K be the set of attributes in the key, and P be the set of attributes appearing in the view. We will assume that the functional dependency KßA is the only consistency constraint on R.

Observe that since the relation may only have a key constraint, the database is in Boyce-Codd Normal Form. We have already dealt with selections. This work differs from prior work on projections [Cosmadakis 83], in that we consider algorithms for translation of view updates. Later in this chapter, we will consider select and project views. In the next chapter, we will consider joins and select, project, and join views. Each view selects attributes P in the relation R, but no selection is performed. In SQL, this would be written as follows:

```
Select P
From R
```

The where clause, were there one, is "true."

We call the attributes appearing in the view the *projected attributes*, while those not appearing in the view are *projected out*. We require that all the attributes of the key must appear in the view (none may be projected out). Any or all of the selecting attributes may be projected out, except for those in the key. This means that the key of the database is the key of the view.

5.2.1 Attribute Independence

Let us define a requirement for the constraints imposed on the database. This requirement will allow attributes of a relation to be updated independently of other attributes, which is useful for updates in the through projections.

We will require the condition *attribute independence*. The attributes of a relation are independently if, given a tuple whose values are arbitrarily chosen from their respective domains, there exists some database extension containing that tuple which satisfies the required constraints.

This condition is similar to, but stronger than, the proscription against *Explicit Functional Dependencies* (EFD) [Cosmadakis 83]. Exclusion of a single combination of attribute values violates attribute independence while not being an EFD.

Let us consider an example of a relation that does not exhibit attribute independence. (This particular relation does so by having EFDs.)

Order Relation

Item	Unit Price	Quantity	Total Price
I	U	Q	T

(We will use ß for FD and $\overset{e}{\text{ß}}$ for EFD.)

Now $IUQ\text{ß}T$, but also $UQ \overset{e}{\text{ß}} T$, $UT \overset{e}{\text{ß}} Q$, and $TQ \overset{e}{\text{ß}} U$. The explicit functional dependency is $U \times Q = T$. The last three mean that we do not have attribute independence. The effect is that we cannot update the value of U in one tuple without also updating the value of Q or T (or both).

By requiring attribute independence, we allow any attribute to be changed independently of any other attribute. We suggest that the database be reorganized to accomplish attribute independence. If an EFD holds, remove attributes until the EFD doesn't hold. Then put the removed attributes in a view as non-updatable attributes whose values are computed based on functions. The removed attribute can be regenerated as *derived data* [Wiederhold 83].

Attribute independence allows the attributes appearing in the view to be changed according to the view update with minimal consideration of the values of the other attributes. When there is an EFD, the view update mechanism may have to compute the value of some attribute not appearing the view based on values that do appear in the view. As an EFD can be arbitrarily complex, the view update mechanism must provide for the description of the EFD in a programming language with the power of a Turing machine. When attribute independence does not hold, but neither does an EFD, particular combinations of attribute values may be proscribed, and the replacement of some value in some view tuple may require some change to some value not appearing in the view because the combination is not permitted. The choice of this other new value is also of arbitrary complexity. When attribute independence does not hold (but neither do EFDs), we will choose to decide the update independently of these additional constraints, but then validate that the choice satisfies all the constraints. If it does not, we will reject the view update, and not attempt to determine more complex alternatives that would satisfy all the constraints.

5.2.2 Case I. Projected attributes include key

We will consider single tuple view updates where the view is a projection of a BCNF relation, R, that includes the key of the relation. Let the key of the relation be represented by X (for one or more attributes). The non-key attributes appearing in the view will be represented by Y, while the non-key attributes not appearing in the view will be represented by Z. The relation has the functional dependency XßYZ. The view is of attributes XY and view updates must satisfy the functional dependency XßY if Y is non-empty. This view has a minimal complement XZ. Our update algorithms do not necessarily hold this complement constant; when the complement is not held constant, the view update would otherwise have to be rejected.

We will now consider deletion of a single view tuple.

Algorithm P-I-D: Delete the database tuple whose key matches the key of the view tuple to be deleted. (That is, perform the deletion request specified against the view directly on the database.)

Lemma 5.10 *The updates generated by Algorithm P-I-D satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuple affected is the deleted view tuple. (2) There is only one database request. (3) The only database request is an deletion, so clearly doing less could not implement the view update request. (4) There are no database replacement requests. (5) There are no database insertion requests. ∎

Theorem 5.4 *The set of update translations that satisfy the five criteria for candidate update translations for individual view deletions are precisely those that are generated by Algorithm P-I-D.*

Proof: The previous lemma shows that the translations generated by Algorithm P-I-D satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that any database update request that implements the view update request must include a deletion or replacement. This is because a database insertion cannot cause a deletion from a monotonic view. The functional dependency forces there to be only one database tuple with a key matching the view tuple. Criterion 1 requires that this be the only tuple affected by the view update. Let us consider the sole database operation that affected this tuple. If it is a deletion, then by Criterion 3, the translation is that generated by Algorithm P-I-D. If is a replacement, we consider whether the key has changed. If it has, this violates Criterion 1. If it has not, the view tuple still appears in the view as the key matches. Consequently, this is not a valid view update translation as it does not perform the deletion completely. ∎

We will now consider insertion of a single view tuple.

Algorithm class P-I-I: Insert a database tuple whose attributes match those of the view tuple to be inserted. The other attributes may be chosen arbitrarily from their respective domains.

Lemma 5.11 *The updates generated by Algorithm class P-I-I satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuple affected is the inserted view tuple. (2) There is only one database request. (3) The only database request is an insertion, so clearly doing less could not implement the view update request. (4) There are no database replacement requests. (5) There are no database deletion requests. ∎

Theorem 5.5 *The set of update translations that satisfy the five criteria for candidate update translations for individual view insertions are precisely those that are generated by Algorithm class P-I-I.*

Proof: The previous lemma shows that the translations generated by Algorithm class P-I-I satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that any database update request that implements the view update request must include an insertion or replacement. This is because a database deletion cannot cause an insertion into a projection view. The functional dependency forces there to be only one database tuple with a key matching the view tuple to be inserted. Criterion 1 requires that this be the only tuple affected by the view update. Let us consider the sole database operation that affects this tuple. If it is an insertion, then by Criterion 3, the translation is that generated by Algorithm class P-I-I. If is a replacement, we consider whether the key has changed. If it has, this violates Criterion 1. If it has not, the view tuple insertion was invalid as it would violate a functional dependency. ∎

We will now consider replacement of a single view tuple.

Algorithm P-I-R: Replace the database tuple whose key matches key of view tuple replaced by changing attributes that change in view tuple as specified, and keeping other attributes constant.

Lemma 5.12 *The updates generated by Algorithm P-I-R satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the replaced view tuple and the replacement view tuple. (2) There is only one database request. (3) There is only one database request, so clearly doing less could not implement the view update request. (4) The single database replacement request changes the key only if the key in the view tuple changes. Other attributes are changed only in the database tuple if they change in the view tuple. Thus fewer attributes cannot be changed. (5) There are neither database deletion nor database insertion requests. ∎

Theorem 5.6 *The set of update translations that satisfy the five criteria for candidate update translations for individual view replacements are precisely those that are generated by Algorithm P-I-R.*

Proof: The previous lemma shows that the translations generated by Algorithm P-I-R satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that the original view tuple must disappear from the view and the new view tuple must now appear in the view. We will first consider the case where the key of the view tuple does not change. This requires a deletion and an insertion, although one of both of these may be substituted by a replacement. Let us first consider the case where the original view tuple was removed by a replacement. Since it is possible to replace the original view tuple (in the database) with the new view tuple that has a matching key, the replacement database tuple must contain the replacement view tuple. By Criterion 4, this replacement must be the one generated by Algorithm P-I-R. A similar argument will show that if the replacement tuple was inserted by a replacement operation, the replaced tuple must have been the database tuple containing the original view tuple. The final alternative is the pair of a deletion and an insertion, which is equivalent to (and, by Criterion 5, can be simplified to) the replacement operation generated by Algorithm P-I-R.

Now, let us consider the case where the key of the view tuple changes. The replaced view tuple corresponds with a database tuple that must be deleted or replaced. Again, we will consider the case where it is replaced. If it is replaced by a tuple with the same key, then the view tuple was not properly replaced, as another tuple with the same key has appeared in its stead. Thus the replacement database tuple must have a different key. That key must be the key of the replacement view tuple, by Criterion 1. The attributes changed must be precisely those changed in the view tuple, by Criterion 4. This is the view update generated by Algorithm P-I-R. We can use the same argument if the replacement view tuple corresponded to a database tuple that was inserted by a replacement operation. The final alternative is a pair of database deletion and insertion operations. This is equivalent to (and, by Criterion 5, can be simplified to) the replacement generated by Criterion 4.

There is one difference between the deletion-insertion pair and the replacement: In the deletion-insertion case, arbitrary attribute values may be used for the attributes not appearing in the view. In the replacement case, the values are taken from the original database tuple. However, since the deletion-insertion pair can be simplified to the replacement, not reusing the attributes would violate Criteria 4 and 5. ∎

Note that each view tuple corresponds to a unique database tuple. These are clearly unique ways to perform the update. Delete and Insert do not preserve any complement. Replace preserves a complement *iff* key does not change.

5.2.3 Adapting Criterion 1 When the Key is not Included in the View

For Cases II and III, where the key is not included in the view, we have to adapt Criterion 1, which requires that the database tuples affected must match the key of some view tuple mentioned in the view update. For these two cases, we will require that the database tuples affected must match all attributes of some view tuple mentioned in the view update. With this revised criterion, called Criterion $1'$, we will be able to discuss the two remaining cases.

5.2.4 Case II. Projected attributes are disjoint from key

We will consider single tuple view updates where the view is a projection of a BCNF relation, R, that does not include the key of the relation. Let the key of the relation be represented by X (for one or more attributes). The non-key attributes appearing in the view will be represented by Y, while the non-key attributes not appearing in the view will be represented by Z. Note that X and Y may not be empty, while Z may be empty. The relation has the functional dependency XßYZ. The view is of attributes Y and view updates do not need to satisfy any functional dependency. (Translations need to satisfy the FD XßYZ, however.) The only complement to this view in SP is XYZ (the identity view). Our update algorithms do not hold this complement constant; when the complement is not held constant, all view updates would otherwise have to be rejected.

We will now consider deletion of a single view tuple.

Algorithm P-II-D: Delete all database tuples which match the view tuple to be deleted in all attributes appearing in the view. (That is, perform the deletion request specified against the view directly on the database.)

Lemma 5.13 *The updates generated by Algorithm P-II-D satisfy the five criteria for candidate update translations.*

Proof: ($1'$) All database tuples affected match the view tuple to be deleted. (2) There are only deletion requests, each mentioning a unique database tuple. (3) If any database deletion request were not done, the view tuple to be deleted would still appear in the view. (4) There are no database replacement requests. (5) There are no database insertion requests. ∎

Theorem 5.7 *The set of update translations that satisfy the five criteria for candidate update translations for individual view deletions are precisely those that are generated by Algorithm P-II-D.*

Proof: The previous lemma shows that the translations generated by Algorithm P-II-D satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that any database update request that implements the view update request must include a deletion or replacement. This is because a database insertion cannot cause a deletion from a monotonic view. Criterion $1'$ requires that only database tuples exactly matching the affected view tuples may be updates. This is the set of database tuples deleted by Algorithm P-II-D. None of these tuples may appear in the database after the update, lest the view tuple to be deleted still appear in the view. For each of these database tuples, let us consider the sole database operation that affected this tuple. If it is a deletion, then by Criterion 3, this part of the translation is that generated by Algorithm P-II-D. If is a replacement, then by Criterion $1'$, the new database tuple must have the same attribute values of an affected view tuple. However, that would imply that the new database tuple would cause the view tuple to be deleted to still appear in the view. Therefore a database replacement cannot be part of this view update translation. Additional deletions are unnecessary and proscribed by Criterion 3. Thus, the only valid view update translation satisfying our criteria deletes all matching database tuples from the view, as generated by Algorithm P-II-D. ∎

We will now consider insertion of a single view tuple.

Algorithm class P-II-I: Insert a database tuple whose attributes match those of the view tuple to be inserted. The other attributes may be chosen arbitrarily from their respective domains, except that the new key is unique.

Lemma 5.14 *The updates generated by Algorithm class P-II-I satisfy the five criteria for candidate update translations.*

Proof: ($1'$) The only tuple affected is the inserted view tuple. (2) There is only one database request. (3) The only database request is an insertion, so clearly doing less could not implement the view update request. (4) There are no database replacement requests. (5) There are no database deletion requests. ∎

Theorem 5.8 *The set of update translations that satisfy the five criteria for candidate update translations for individual view insertions are precisely those that are generated by Algorithm class P-II-I.*

Proof: The previous lemma shows that the translations generated by Algorithm class P-II-I satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that any database update request that implements the view update request must include an insertion or replacement. This is because a database deletion cannot cause an insertion into a projection view. A database replacement would affect a database tuple which does not match the attribute values of an affected view tuple. (If there were a database tuple that matched the view tuple to be inserted, the view tuple could not be inserted as it would already be there.) Therefore, the view tuple insertion must be implemented by at least one database insertion. We observe that the view tuple to be inserted does not match any existing view tuple. This implies that the key of this new tuple must not match any existing tuple, as it would violate the key (functional) dependency. Therefore, the tuple must be inserted with a unique key, but all other attributes may be arbitrarily chosen. ∎

We will now consider replacement of a single view tuple.

Algorithm class P-II-R: Replace all database tuples which exactly match all attributes of the view tuple replaced by changing attributes that change in view tuple as specified, and keeping other attributes constant. Some but not all of the replacements may be deletions instead, provided that there is at least one replacement generating the desired replacement view tuple.

Lemma 5.15 *The updates generated by Algorithm class P-II-R satisfy the five criteria for candidate update translations.*

Proof: $(1')$ The only tuples affected are the database tuples matching the replaced view tuple and the replacement view tuple. (2) Each database request affects a different original database tuple. (3) If not all the replacements or deletions were done, the original view tuple (to be replaced) would still appear in the view. (4) None of the database tuples have their key changed. Other attributes are changed only in the database tuple if they change in the view tuple. Thus fewer attributes cannot be changed. (5) There are no database insertion requests. ∎

Theorem 5.9 *The set of update translations that satisfy the five criteria for candidate update translations for individual view replacements are precisely those that are generated by Algorithm class P-II-R.*

Proof: The previous lemma shows that the translations generated by Algorithm class P-II-R satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that the original view tuple must disappear from the view and the new view tuple must now appear in the view. This requires that all database tuples matching the replaced view tuple must disappear from the database, and a database tuple matching the replacement view tuple must be added to the database. We can consider how the database tuples to be removed actually got removed (by either deletion or replacement), and how the database tuple(s) to be added to the database (by either insertion or replacement). If the database tuple(s) to be added were replacement tuples, they must be replacement tuples of database tuples to be deleted, by Criterion $1'$. Also by Criterion $1'$, if the database tuples to be removed were replaced, the replacement tuples must be the replacement tuples to be added. Thus, the other tuples to be removed must have been deleted, and the other tuples to be added must have been inserted. We only need one insertion (by Criterion 3), and can be combined with a deletion to make a replacement, as required by Criterion 5 and as generated by Algorithm class P-II-R. Therefore, all translations satisfying our five criteria were generated by Algorithm class P-II-R. ∎

5.2.5 Case III. Projected attributes include only partial key

We will consider single tuple view updates where the view is a projection of a BCNF relation, R, that overlaps the key of the relation. Let the part of the key of the relation appearing in the view be represented by X (for one or more attributes), while the part of the key of the relation not appearing in the view be represented by W. The non-key attributes appearing in the view will be represented by Y, while the non-key attributes not appearing in the view will be represented by Z. Note that W and X may not be empty, while either or both of Y and Z may be empty. The relation has the functional dependency WXßYZ. The view is of attributes XY and view updates do not need to satisfy any functional dependency. (Translations need to satisfy the FD WXßYZ, however.) The only complement to this view in SP is WXYZ (the identity view). Our update algorithms do not hold this complement constant; when the complement is not held constant, all view updates would otherwise have to be rejected.

We will now consider deletion of a single view tuple.

Algorithm P-III-D: Delete all database tuples which match the view tuple to be deleted in all attributes appearing in the view. (That is, perform the deletion request specified against the view directly on the database.)

Lemma 5.16 *The updates generated by Algorithm P-III-D satisfy the five criteria for candidate update translations.*

Proof: ($1'$) All database tuples affected match the view tuple to be deleted. (2) There are only deletion requests, each mentioning a unique database tuple. (3) If any database deletion request were not done, the view tuple to be deleted would still appear in the view. (4) There are no database replacement requests. (5) There are no database insertion requests. ∎

Theorem 5.10 *The set of update translations that satisfy the five criteria for candidate update translations for individual view deletions are precisely those that are generated by Algorithm P-III-D.*

Proof: The previous lemma shows that the translations generated by Algorithm P-III-D satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that any database update request that implements the view update request must include a deletion or replacement. This is because a database insertion cannot cause a deletion from a monotonic view. Criterion $1'$ requires that only database tuples exactly matching the affected view tuples may be updates. This is the set of database tuples deleted by Algorithm P-III-D. None of these tuples may appear in the database after the update, lest the view tuple to be deleted still appear in the view. For each of these database tuples, let us consider the sole database operation that affected this tuple. If it is a deletion, then by Criterion 3, this part of the translation is that generated by Algorithm P-III-D. If is a replacement, then by Criterion $1'$, the new database tuple must have the same attribute values of an affected view tuple. However, that would imply that the new database tuple would cause the view tuple to be deleted to still appear in the view. Therefore a database replacement cannot be part of this view update translation. Additional deletions are unnecessary and proscribed by Criterion 3. Thus, the only valid view update translation satisfying our criteria deletes all matching database tuples from the view, as generated by Algorithm P-III-D. ∎

We will now consider insertion of a single view tuple.

Algorithm class P-III-I: Insert a database tuple whose attributes match those of the view tuple to be inserted. The other attributes may be chosen arbitrarily from their respective domains, except that the remainder of the key must be chosen so that the new key is unique.

Lemma 5.17 *The updates generated by Algorithm class P-III-I satisfy the five criteria for candidate update translations.*

Proof: (1′) The only tuple affected is the inserted view tuple. (2) There is only one database request. (3) The only database request is an insertion, so clearly doing less could not implement the view update request. (4) There are no database replacement requests. (5) There are no database deletion requests. ∎

Theorem 5.11 *The set of update translations that satisfy the five criteria for candidate update translations for individual view insertions are precisely those that are generated by Algorithm class P-III-I.*

Proof: The previous lemma shows that the translations generated by Algorithm class P-III-I satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that any database update request that implements the view update request must include an insertion or replacement. This is because a database deletion cannot cause an insertion into a projection view. A database replacement would affect a database tuple which does not match the attribute values of an affected view tuple. (If there were a database tuple that matched the view tuple to be inserted, the view tuple could not be inserted as it would already be there.) Therefore, the view tuple insertion must be implemented by at least one database insertion. We observe that the view tuple to be inserted does not match any existing view tuple. This implies that the key of this new tuple must not match any existing tuple, as it would violate the key (functional) dependency. Therefore, the tuple must be inserted with a unique key, but all other attributes may be arbitrarily chosen. ∎

We will now consider replacement of a single view tuple.

Algorithm class P-III-R: Replace all database tuples which exactly match all attributes of the view tuple replaced by changing attributes that change in view tuple as specified, and keeping other attributes constant. Some but not all of the replacements may be deletions instead, provided that there is at least one replacement generating the desired replacement view tuple. If the partial key of the view tuple changes, an FD conflict may arise between some new (replacement) database tuple and an existing database tuple. In this case, we make an arbitrary change to some attribute of the part of the key not appearing in the view to avoid the FD conflict.

Lemma 5.18 *The updates generated by Algorithm P-III-R satisfy the five criteria for candidate update translations.*

Proof: (1′) The only tuples affected are the database tuples matching the replaced view tuple and the replacement view tuple. (2) Each database request affects a different original database tuple. (3) If not all the replacements or deletions were done, the original view tuple (to be replaced) would still appear in the view. (4) Database tuples have their key changed only if due to an FD conflict, in which case, either the key must be changed or the conflicting tuple must be updated. The latter alternative violates Criterion I′. Other attributes are changed only in the database tuple if they change in the view tuple. Thus fewer attributes cannot be changed. (5) There are no database insertion requests. ∎

Theorem 5.12 *The set of update translations that satisfy the five criteria for candidate update translations for individual view replacements are precisely those that are generated by Algorithm P-III-R.*

Proof: The previous lemma shows that the translations generated by Algorithm P-III-R satisfy the five criteria. We need to show that no other translations satisfy these five criteria.

We first observe that the original view tuple must disappear from the view and the new view tuple must now appear in the view. This requires that all database tuples matching the replaced view tuple must disappear from the database, and a database tuple matching the replacement view tuple must be added to the database. We can consider how the database tuples to be removed actually got removed (by either deletion or replacement), and how the database tuple(s) to be added to the database (by either insertion or replacement). If the database tuple(s) to be added were replacement tuples, they must be replacement tuples of database tuples to be deleted, by Criterion $1'$. Also by Criterion $1'$, if the database tuples to be removed were replaced, the replacement tuples must be the replacement tuples to be added. Thus, the other tuples to be removed must have been deleted, and the other tuples to be added must have been inserted. We only need one insertion (by Criterion 3), and can be combined with a deletion to make a replacement, as required by Criterion 5 and as generated by Algorithm P-III-R. Therefore, all translations satisfying our five criteria were generated by Algorithm P-III-R. ∎

Case III is very similar to Case II. Algorithm P-III-D is exactly the same as Algorithm P-II-D. Algorithm class P-III-I is changed from Algorithm class P-II-I only in that a smaller portion of the key is available to generate a unique key. Algorithm P-III-R has an additional consideration not found in Algorithm class P-II-R: the possibility of an FD conflict, which requires changing the key.

5.2.6 About the three cases

Case I—projection includes key—is a good problem with a clear solution. As we have seen, the other cases, where the projection does not include the entire key, involve generation of (partial) keys on insertion, a rather undesirable property. We have included these latter two cases for completeness, but recommend against using these algorithms. Rather, we recommend that the view always contain the key, which permits straightforward translations of view updates.

This work on project operations is part of work on select-project-join views. Our model for joins requires that the key be present in a root relation. The joins are all extension joins cascading from the root, and all the join attributes must be present in the view (see the next chapter). Therefore, all the keys to each BCNF relation appears in the view. Consequently, Case I suffices.

Operation	Key appears (Case I)	Key does not appear (Case II)	Partial key appears (Case III)
Delete	Delete matching DB tuple	Delete all matching DB tuples	Delete all matching DB tuples
Insert	Extend view tuple with arbitrary values and insert into DB	Extend view tuple with arbitrary values and unique key and insert into DB	Extend view tuple with arbitrary values so key is unique and insert into DB
Replace	Change matching DB tuple as specified in view update	Change at least one matching DB tuple as specified in view update, can delete others if desired	Change at least one matching DB tuple as specified in view update, changing part of key not appearing in the view if FD conflict arises from changing part of key appearing in view, can delete others if desired

Summary of Algorithms for Projections

5.3 Selections and Projections

We will deal with select/project views on a single relation with a single key dependency. Let R be the set of attributes in the relation **R**, and let K be the set of attributes in the key. We will assume that the functional dependency KßR is the *only* consistency constraint on **R**. Observe that since the relation may only have a key constraint, the database has already undergone normalization.

Let us first consider the selection condition. The selection condition is a (possibly empty) conjunction of terms of the form $A \in s$ (or equivalently, $A \notin e$), where s (and e) is a set of constants in the domain of A. We call the values in s *selecting values*, and the values in e *excluding values*. For non-selecting attributes the set of selecting values is the entire domain and the set of excluding values is the empty set. We call the attributes appearing in the selection condition *selecting attributes*. If the selection condition is "true" (i.e., an empty conjunction), the set of selecting attributes is empty, the sets of selecting values are the entire domains, and the sets of excluding values are empty. This type of selection condition allows attributes to be treated independently in view updates.

We call the attributes appearing in the view the *projected attributes*, while those not appearing in the view are *projected out*. We require that all the attributes of the key must appear in the view (none may be projected out). Any or all of the selecting attributes may be projected out, except for those in the key. This means that the key of the database is the key of the view.

5.3.1 Translation of Insertion Requests

The request is to insert a single, fully-specified view tuple. The new view tuple must be a valid view tuple—it must satisfy the selection condition—and not conflict with any existing view tuple. That is, there must not already be a tuple in the view with the key of the view tuple to be inserted. The extend-insert algorithms are subroutines of Algorithms classes I-1 and I-2, which are the algorithms that translate view insertions.

The algorithms for select/project views are combinations of the algorithms for select views and for project views. Algorithm class I-1 is the combination of algorithm S-I-1 and algorithm class P-I-I. Algorithm class I-2 is the combination of algorithm S-I-2 and algorithm class P-I-I. The former handles insertions where there is no key conflict in the database, the latter handles insertions where there is a key conflict.

Algorithm class extend-insert: The new database tuple is formed by taking the attributes from the new view tuple as supplied. For remaining attributes, the values are chosen arbitrarily from their respective sets of selecting values (which is the domain for non-selecting attributes). Each combination of values represents a different algorithm from this class. There is a unique extend-insert algorithm iff each attribute projected out has set of selecting values that is a singleton (has only one element).

Algorithm class I-1: If the new view tuple does not conflict with (have the same key as) any existing *database* tuple, then insert the tuple obtained by one of the extend-insert algorithms, else reject the update request. There is an algorithm in class I-1 for each extend-insert algorithm.

Lemma 5.19 *The updates generated by Algorithm class I-1 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuple affected is the inserted view tuple. (2) There is only one database request. (3) The only database request is an insertion, so clearly doing less could not implement the view update request. (4) There are no database replacement requests. (5) There are no database deletion requests. ∎

Algorithm class I-2: If the new view tuple has a key matching that of an existing database tuple, then change the attributes in the database tuple to match the new view tuple and change all attributes in the database tuple with excluding values to arbitrary selecting values. There is an algorithm in class I-2 for every combination of one selecting value from zero or more selecting attributes other than the key.

Lemma 5.20 *The updates generated by Algorithm class I-2 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the inserted view tuple and the replaced database tuple, both with the same key value. (2) There is only one database request. (3) The only database request is a single replacement, so clearly doing less could not implement the view update request. (4) There is a single database replacement request, and it changes only those attributes that differ between the old database tuple and the new view tuple. No replacement could be simpler because there are no other candidate tuples for replacement with the same key. (5) There are no database deletion requests. ∎

Theorem 5.13 *The set of update translations that satisfy the 5 criteria for candidate update translations for individual view insertions are precisely those in algorithm classes I-1 and I-2.*

Proof: The two previous lemmata show that the translations generated by the two algorithms satisfy the five criteria.

We will now show that there are no other translations satisfying the five criteria. We first observe that every view update valid in the view can be translated by either Algorithm class I-1 or by Algorithm class I-2. Because the view is monotonic—a view is monotonic if view formation preserves inclusion (i.e., $A \subseteq B ßv(A) \subseteq v(B)$)—an view insertion cannot be translated solely by database deletions. Therefore, any set of database updates implementing the view insertion request must contain at least one insertion or replacement. Precisely one of these requests must refer to the new view tuple. If it is a replacement, the replacement tuple must be the new view tuple as no subsequent database request can affect that tuple in this view update request. That request is generated by Algorithm class I-2, as the replaced tuple must have a matching key. If it is an insertion, the tuple inserted must be the new view tuple as no subsequent database request can affect that tuple in this view update request. That request is generated by Algorithm class I-1. However, by Criterion 5, the translation may not separately request deletion of a conflicting database tuple—the one replaced in Algorithm class I-2. Such a sequence would be simplified to the acceptable translation from Algorithm class I-2. ▮

Let us consider when these translations may be used. There may be several translations in algorithm class I-1 that may be applicable at the same time (although only one should be chosen). Similarly for algorithm class I-2. However, the translations in algorithm class I-1 apply to a disjoint set of database states from the translations in algorithm class I-2. In particular, a database state has at least one valid translation from algorithm class I-1 or from algorithm class I-2 *but not both*. Note that there are translators which are formed by combinations of translations of algorithms in classes I-1 and I-2, including those which can translate all legal view update requests.

5.3.2 Translation of Deletion Requests

The request is to delete a single, fully-specified view tuple. The deleted view tuple must currently be in the view.

The algorithms for select/project views are combinations of the algorithms for select views and for project views. Algorithm class D-1 is the combination of algorithm S-D-1 and algorithm class P-I-D. Algorithm class D-2 is the combination of algorithm S-D-2 and algorithm class P-I-D. Once again, algorithm classes D-1 and D-2 are the analogs of algorithm classes I-1 and I-2, respectively.

Algorithm class D-1: Delete the database tuple whose key matches that of the view tuple to be deleted. There is only one algorithm in class D-1 for each view update request.

Lemma 5.21 *The updates generated by Algorithm class D-1 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuple affected is the deleted view tuple. (2) There is only one database request. (3) The only database request is an deletion, so clearly doing less could not implement the view update request. (4) There are no database replacement requests. (5) There are no database insertion requests. ∎

Algorithm class D-2: Replace the database tuple whose key matches that of the view tuple to be deleted, changing one non-key selecting attribute to an arbitrary excluded value. There is an algorithm in class D-2 for every non-key excluding value. Of course, there is no algorithm in class D-2 if the selection condition is "true" (i.e., there is no select clause) or if the set of selecting attributes is a subset of the key.

Lemma 5.22 *The updates generated by Algorithm class D-2 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the deleted view tuple and the replacement database tuple (whose key matches that of the deleted view tuple). (2) There is only one database request. (3) The only database request is a single replacement, so clearly doing less could not implement the view update request. (4) The key is not changed. There is a single database replacement request, and it changes only those attributes that differ between the old database tuple and the new view tuple. No replacement could be simpler because there are only one (non-key) attribute is changed. (5) There are no database insertion requests. ∎

Theorem 5.14 *The set of update translations that satisfy the 5 criteria for candidate update translations for individual view deletions are precisely those in algorithm classes D-1 and D-2.*

Proof: The two previous lemmata show that the translations generated by the two algorithms satisfy the five criteria.

We will now show that there are no other translations satisfying the five criteria. We first observe that every view update valid in the view can be translated Algorithm class D-1 and sometimes by Algorithm class D-2. Because the view is monotonic, a view deletion cannot be translated solely by database insertions. Therefore, any set of database updates implementing the view deletion request must contain at least one deletion or replacement. Precisely one of these requests must refer to the deleted view tuple. If it is a replacement, the replaced tuple must be the deleted view tuple, as no preceding database request can affect that tuple in this view update request by Criterion 2. That request is generated by Algorithm class D-2, and the replacement tuple has a matching key if possible. If it is an deletion, the tuple deleted must be the deleted view tuple as no preceding database request can affect that tuple in this view update request. That request is generated by Algorithm class D-1. ∎

Let us consider when these translations may be used. The single algorithm in class D-1 is always applicable. The algorithms in class D-2 are applicable when they exist. One can consider algorithm class D-1 to be the inverse of algorithm class I-1, and algorithm class D-2 to be the inverse of algorithm class I-2. We note that these inverses are not perfect, which is why the insertion and deletion of the same tuple (or vice versa) is not necessary. Furthermore, there is no analog for deletion of an translator that combines algorithms from classes I-1 and I-2.

5.3.3 Translation of Replacement Requests

The request is to replace a single, fully-specified view tuple with another. The replaced view tuple must currently be in the view, and the replacement view tuple must currently not be in the view. Both tuples must satisfy the selection condition. It must be possible to do the replacement in the view; that is, if there is a tuple in the view whose key matches that of the replacement tuple, it must be the replaced tuple.

Algorithm extend-replace: Replace the database tuple changing the attributes appearing in the view to match those in the new view tuple. When used in algorithm classes R-1 and R-2, this will mean changing only those attributes that change in the view tuple replacement. This is similar to algorithm class I-2, except that the replaced database tuple *does* appear in the view. There is only one extend-replace algorithm.

Algorithm class R-1: If the key does not change in the view tuple replacement, then perform algorithm extend-replace changing only those attributes that change in the view tuple replacement. Otherwise, reject the update request.

Lemma 5.23 *The updates generated by Algorithm class R-1 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the replaced and replacement view tuple. (2) There is only one database request. (3) The only database request is a single replacement, so clearly doing less could not implement the view update request. (4) There is a single database replacement request, and it changes only those attributes that differ between the old view tuple and the new view tuple. No replacement could be simpler because there are no other candidate tuples for replacement with the same key. (5) There are neither database insertion nor database deletion requests. ∎

Note that if the key does not change, there is no possibility of a conflict with any tuple not appearing in the view.

Algorithms R-2 through R-5 handle the case where the key changes in the view update request, and are summarized in the chart.

Algorithm class R-2: Perform algorithm extend-replace if the key changes in the view tuple replacement and there is no tuple in the database whose key matches that of the replacement view tuple. Otherwise, reject the update request.

Lemma 5.24 *The updates generated by Algorithm class R-2 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the replaced view tuple and the replacement view tuple. (2) There is only one database request. (3) The only database request is a single replacement, so clearly doing less could not implement the view update request. (4) There is a single database replacement request, and it changes only those attributes that differ between the old database tuple and the new view tuple. No replacement could be simpler because there are no candidate tuples for replacement with the same key. (It is possible to use a different replacement tuple with the same key, but that would require an additional database request, as we will see in Algorithm class R-4.) (5) There are neither database insertion nor database deletion requests. ∎

Algorithm class R-2 will not allow changes to database tuples not appearing in the view.

Algorithm class R-3: If the key changes in the view tuple replacement and there *is* a tuple in the database whose key matches that of the replacement view tuple, then perform an algorithm of class I-2 (on the new view tuple) and delete the database tuple whose key matches that of the replaced view tuple. Otherwise, reject the update request.

Lemma 5.25 *The updates generated by Algorithm class R-3 satisfy the five criteria for candidate update translations.*

Proof: (1) The database update is the deletion of the original view tuple and the replacement of a database tuple not appearing in the view with the new view tuple. (2) The three changed database tuples are all distinct. (3) All three tuples must be involved in the database update for the update to be translated. This is because the replaced view tuple must not appear in the view (so it must not appear in the database), the replacement view tuple must appear in the view (so it must now appear in the database), and the conflicting database tuple cannot be in the database along with the replacement view tuple (so something must be done to it also). At least two requests are necessary to affect three tuples, and there are exactly two requests. (4) The single database replacement request does not change the key. It can only be simplified by changing fewer attributes. However, there are no alternative replaced or replacement tuples with the same key. Therefore, the replacement cannot be simplified. (5) There are no database insertion requests. ∎

Algorithm class R-3 changes a database tuple that does not appear in the view (because otherwise the view tuple replacement is not valid). If we consider the view replacement as a view deletion and a view insertion, then the dichotomy between Algorithm classes I-1 and I-2 parallels that between Algorithm classes R-2 and R-3 (respectively).

Algorithm classes R-2 and R-3 assume that no tuple remains in the database with a key matching that of the replaced view tuple. However, it is possible to replace (in the database) the replaced view tuple with a tuple with a matching key that does not satisfy the selection criteria. This parallels the dichotomy of Algorithm classes D-1 and D-2 for deletion. We obtain two algorithm classes using the same conditions of Algorithm classes R-2 and R-3, respectively.

Algorithm class R-4: If the key changes in the view tuple replacement and there is no tuple in the database whose key matches that of the replacement view tuple,

Delete / Insert	Delete replaced tuple (cf. algorithm D-1)	Replace (in database) replaced view tuple (cf. algorithm D-2)
No conflict for replacement view tuple (cf. algorithm I-1)	Algorithm class R-2 One replacement	Algorithm class R-3 Replace old view tuple Insert new view tuple
There is a database tuple whose key matches the replacement view tuple (cf. algorithm class I-2)	Algorithm class R-4 Delete old view tuple Replace new view tuple	Algorithm class R-5 Replace both old and new view tuples

How algorithms for replacement when the key changes are related to deletion and insertion algorithms

perform an algorithm from class D-2 (for the replaced view tuple) and an algorithm from class I-1 (for the replacement view tuple). (That is, the replaced view tuple will be changed to not appear in the view and the replacement view tuple will be inserted.) Otherwise, reject the update request.

Lemma 5.26 *The updates generated by Algorithm class R-4 satisfy the five criteria for candidate update translations.*

Proof: (1) The only tuples affected are the replaced view tuple, a new database tuple with a matching key, and the replacement view tuple. (2) The replacement view tuple (the inserted database tuple) does not match the two tuples mentioned in the database replacement. (3) Neither request is sufficient by itself. (4) The only database replacement request changes only a single non-key attribute. (5) There are no database deletion requests. ∎

Algorithm class R-5: If the key changes in the view tuple replacement and there *is* a tuple in the database whose key matches that of the replacement view tuple, then perform an algorithm from class D-2 (for the replaced view tuple) and an algorithm from class I-2 (for the replacement view tuple). (That is, the replaced view tuple will be changed to not appear in the view and the replacement view tuple will be obtained by replacing some database tuple that did not appear in the view.) Otherwise, reject the update request.

Lemma 5.27 *The updates generated by Algorithm class R-5 satisfy the five criteria for candidate update translations.*

Proof: (1) There four tuples affected in the database update. The original view tuple is replaced by a database tuple with a matching key. The replacement view tuple replaces a database tuple with a matching key. (2) The four database tuples are clearly distinct. (3) Each of the replacements is needed: one removes the replaced view tuple, the other inserts the replacement view tuple. (4) Both database replacements do not change the key. The database replacement that inserts the new view tuple can only be simplified by changing fewer attributes. However, there are no alternative replaced or replacement tuples with the same key. Therefore, that replacement cannot be simplified. The other database replacement request changes only a single non-key attribute. (5) There are neither database deletions nor database insertions. ▮

Algorithm class R-1 is the only one possible when the replacement does not change the key. When the view replacement does change the key, we have two options for handling the replaced view tuple corresponding to Algorithm classes D-1 and D-2, and two situations involving the replacement view tuple corresponding to Algorithm classes I-1 and I-2. The four replacement algorithm classes are described by the table.

Theorem 5.15 *The set of update translations that satisfy the five criteria for candidate update translations for individual view replacements are precisely those that are generated by Algorithm classes R-1, R-2, R-3, R-4, and R-5.*

Proof: The five previous lemmata show that the translations generated by the five algorithms satisfy the five criteria.

We will now show that there are no other translations satisfying the five criteria. We consider the three exclusive cases based on the enabling conditions for Algorithm class R-1, for Algorithm classes R-2 and R-4, and for Algorithm classes R-3 and R-5.

Case 1. The key does not change. (Algorithm class R-1.)

The view update requests replacement of one tuple by another. The replaced tuple must have been in the database and the replacement tuple must now be in the database. Since the two tuples have the same key, the replaced tuple must no longer be in the database. We have already seen that the only database operations that remove a tuple from the database are delete and replace.

If the operation is replace (the tuple to be removed by something else), the request where the replacement tuple is the new view tuple is strictly simpler than all other possible replacements. In that case, the database replacement must be precisely the requested view replacement. No other operations are possible (in addition) as the database replacement is sufficient. This is the translation generated by Algorithm class R-1.

If the operation is delete, we consider how the new view tuple appeared in the database. A new tuple can only appear in a database by insertion or replacement. If it was by insertion, this insertion and (previous) deletion are together sufficient, and the two are equivalent to (and not as simple as, by Criterion 5) the replacement generated by Algorithm class R-1. On the other hand, if the new view tuple got into the database by replacement, an argument similar to that in the previous paragraph will show that the replaced database tuple must have been the replaced view tuple. With the precondition (of this paragraph) that this tuple was deleted (rather than replaced), we have a contradiction by Criterion 2.

Case 2. The key changes, but there is no (existing) database tuple whose key matches that of the new view tuple. (Algorithm classes R-2 and R-4.)

Let us consider what happens to the replaced view in the database. It can either be deleted or replaced. If it is deleted, we consider how the replacement view tuple got in the database. It could have been inserted or it could be a replacement. If that tuple was inserted, then the deletion and insertion are equivalent to (not as simple as) the replacement generated by Algorithm class R-2. If it is a replacement, it must have replaced a database tuple with a differing key. By Criterion 1, the database tuple replaced must be the view tuple replaced. With the assumption that this tuple was deleted (rather than replaced), we have a contradiction by Criterion 2.

Alternatively, the replaced view tuple is replaced in the database. The replacement database tuple can have a matching or non-matching key. If a non-matching key, by Criterion 1, the replacement database tuple must be the replacement view tuple resulting in Algorithm class R-2. If the database replacement does not change the key, the simplest replacements change only one attribute, as in Algorithm class R-4. In this case, we must also consider how the replacement view tuple got in the database. Again, it could have been inserted or it could be a replacement. If that tuple was inserted, then the replacement and insert is as generated by Algorithm class R-4. If it is a replacement, it must have replaced a database tuple with a differing key. By Criterion 1, the database tuple replaced must be the view tuple replaced. However, we assumed that the original view tuple was replaced by a tuple with a matching key. Consequently, we have a contradiction by Criterion 2.

Case 3. The key changes, and there was a database tuple (that did not appear in the view) whose key matches that of the new view tuple. (Algorithms classes R-3 and R-5.)

Criterion 1 states that affected tuples must have keys that match the tuples mentioned in the view update. The two tuples with the key of the replacement view tuple are the matching database tuple that did not appear in the view and the view tuple itself. There is the possibility that there is a new database tuple (not appearing in the view) whose key matches that of the replaced view tuple. Thus, we have exactly three or four tuples affected by the translation of the view update. If only three tuples are affected, the combination of insertions, deletions, and replacements is equivalent to (and not as simple as) the translations generated by Algorithm class R-3. If four tuples are affected, this extra tuple must be a replacement rather than an insertion (which would be an extraneous operation). It must be a replacement of the replaced view tuple and only one attribute may change as a consequent of Criterion 4. ∎

5.4 Conclusion

We have devised five criteria for acceptable view update translations. We have enumerated a complete list of translators that satisfy these five criteria for a large class of select/project views on Boyce-Codd Normal Form relations. Our techniques take into account the possibility that an object the user has requested to be deleted should actually be transformed into an object the user does not know about, and the possibility that an object the user wants inserted may refer to an existing object the user has just become aware of. Thus an object can be deleted by "destroying" it or converting it into another, unrecognizable object.

With a complete list of alternative translations, we have circumscribed the search space for a translator for view updates (into database updates). Additional semantics is needed to choose the desired translator. Collecting, coding, and using such additional semantics is beyond the scope of this paper.

We handle this large class of select/project views on Boyce-Codd normal form relations. That is, there is a single consistency constraint: a key dependency (or functional dependency). The selection condition is the (possibly empty) conjunction of terms, each of the form $attribute \in set$. The projection may remove any attributes mentioned in the selection condition, except that the key of the relation must appear in the view.

Chapter 6

Updating Views Involving Joins

6.1 Views Consisting of Selections, Projections, and Joins

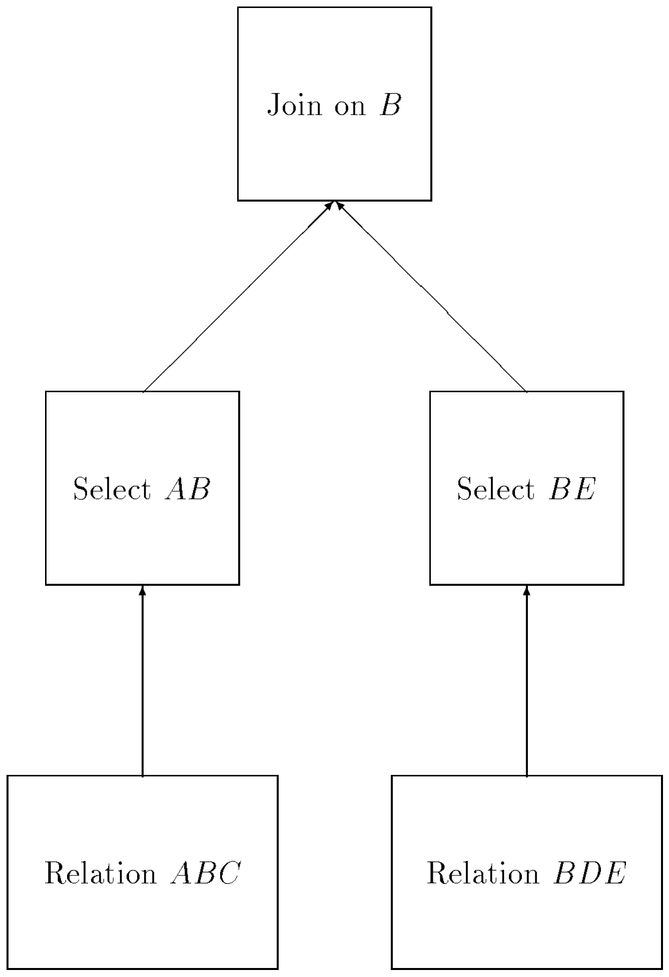
We will now consider views defined using joins as well as selections and projections. We define a query to be in *Select-Project-Join Normal Form* (or *SPJNF*) when it does the selections first, the projections next, and the joins last. Note in particular that this implies that the join attributes must appear in the view.

Theorem 6.1 *Any relational query where no projection removes a join attribute and the selection conditions are conjunctions of the form "attribute in set of constants" can be converted into an equivalent (results in the same answer) relational query that is in SPJNF.*

Proof: Create a directed query graph where each operator and base relation is represented by a node. The edges are directed toward the operators that use them as arguments. Selections and projections have in-degree one. Joins have in-degree two. We shall describe a series of transformations of the graph into one for SPJNF.

Move selection closer to base relation swapping with projection. This transformation does not affect the result as the selection condition can safely ignore the additional attributes and the same attributes appear in the result.

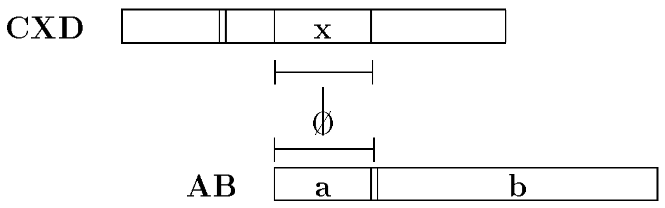
Compose selection conditions of two adjacent selection nodes by *and*ing them. This transformation does not affect the result as the tuple flow through the graph has *and* semantics.



Query Graph

Compose projection lists of two adjacent projection nodes by discarding the one closer to the base relation. The list of attributes emanating from the closer node is a (not necessarily proper) superset of the list of attributes emanating from the farther node. The only attributes appearing out of the pair are those that appear from the farther node.

A join followed by a projection can be replaced by two projections followed by a join where the attributes emanating from each new projection is the intersection of the list entering the old join and the list emanating from the original projection. Since the original projection did not exclude the join attributes, the join is still possible after the new projections. Note that we can discard an identity projection node collapsing



Reference Connection

the paths through it.

A join followed by a selection can be replaced by two selections followed by a join. The two selections list the conditions for the attributes entering the respective branch of the join. Since the two selection conditions are effectively *and*ed together by the join, the original condition must be conjunction.

The result of applying these transformations as often as possible is a query graph in SPJNF. ∎

A view in SPJNF is the composition of a view consisting of joins with some number of select and project views (each on the individual relations), any of which may be the identity. We shall show how to update through the join view and then prove that composing the views works as expected.

6.2 Requirements for Joins

There are two requirements for joins. First, each join must be an extension join with an inclusion dependency. Second, the combination of joins must have a particular pattern.

Each join must be along a reference connection. A reference connection [El-Masri 79, 80, Wiederhold 83] is the combination of an extension join [Honeyman 80] and an inclusion dependency [Casanova 82]. In an extension join, the join attributes are the key of one of the relations. In the figure, the join attributes are **X** in relation **CXD** and **A** in relation **AB**. Notice that **A** is the key for relation **AB**. In addition, we require that for each value **X** in **CXD**, there is a corresponding tuple in **AB** with a matching **A** value.

We can obtain a *query graph* by constructing a graph where each node corresponds to a relation in the view and each edge corresponds to a join in the view definition

[Finkelstein 82]. The edges have directions as shown in the figure (in the many-to-one direction). We shall require that the join views correspond to a query graph that is a tree where all edges are directed away from a single root and each node refers to a unique relation.[1] Note that the key of the root is the key of the entire view.

We will call the class of views in SPJNF where the joins satisfy the two requirements of this section (reference connection and rooted tree), the projections do not remove any key, and the selection conditions are conjunctions of the form $A \in s$ where s is a set of constants in the domain for attribute A.

6.3 Updating Join Views

In this section, we consider the problem of updating join views. Each relation of the join view corresponds to an underlying relation of the database or a select and project view of such a relation. Of course, the SP views could be the identity view (i.e., no selection or projection). Let us first consider how to delete a tuple from the join view where the SP views are identity views, and then extend our results to handle SP views that are not the identity.

Algorithm class SPJ-D: Delete the tuple from the root relation (or SP view) only using one of the algorithm of classes D-1 or D-2.

Theorem 6.2 *Algorithm class SPJ-D is the only algorithm that satisfies the five criteria that deletes a tuple from join views of the type specified when the SP views are identity views.*

Proof: For indentity views, the translation is to delete the corresponding underlying tuple from the root relation. We will first show that this translation performs the deletion without any view side effects and satisfy the five criteria. We will then show that there are no other such translations.

This translation performs the deletion and satisfies the five criteria. Deleting a join view tuple can be accomplished by deleting one of the joined tuples. (1) The only tuple affected matches the projection of the join view tuple. (2) There is only one database request. (3) The only database request is an deletion, so clearly doing less

[1]We can relax this constraint to allow rooted DAGs if we relax the five criteria somewhat.

could not implement the view update request. (4) There are no database replacement requests. (5) There are no database insertion requests. Since the key of the root relation functionally determines the view, the only view tuple that can be affected by deleting the root tuple is the view tuple we want to delete.

Deleting a tuple from a join view can only be performed by deleting one or more of the tuples joined together to form the view tuple. Deleting the root tuple is sufficient. Not deleting the root tuple, but deleting any other of the corresponding underlying tuples would violate the inclusion dependency and has the potential for side effects. Deleting tuples in addition to the root tuple would violate Criterion 3. Thus, we must delete precisely the root corresponding underlying tuple. ∎

We next consider inserting a tuple into the join view. This involves inserting the various projections of the new join view tuple into the individual relations.

Algorithm class SPJ-I: Take the projections of the join view to the attributes listed in each SP view. On each projection (or SP view) there are three cases:

Case 1: The projection exists in the SP view in the exact projected form. If this is the root SP view, reject the update as it violates an FD in the view. Otherwise, we need do nothing with this SP view.

Case 2: The projection does not match the key of any tuple in the SP view. Perform an SP view insertion using the projection of the new join view tuple.

Case 3: There is already a tuple in the SP view with a key matching that of the projection, but the other values do not match. Replace (in the SP view) the existing SP view tuple by the projection of the new join view tuple. We may reject the update request if we do not wish to perform a replacement in the SP view.

If any of the SP view operations fail, the entire view update request fails and is undone.

Theorem 6.3 *Algorithm class SPJ-I is the only algorithm that satisfies the five criteria that inserts a tuple from join views of the type specified when the SP views are identity views.*

Proof: The desired view tuple can only appear in the view if the projections of that view tuple appear in the underlying relations. We will consider each case and show why the action taken is the only possible. In Case 1 doing anything else would violate Criterion 3 as they are unnecessary. In Case 2 the only operations that can cause the desired database tuple (the projection of the view tuple) to appear in the database are insertion and replacement. We have specified insertion of that tuple. If any other database tuple is replaced, it would violate Criterion 1. In Case 3 insertion and replacement are again the only options. Insertion is impossible because of the conflicting tuple. Replacement of any other tuple would violate Criterion 1 Therefore, only the operation specified in Case 3 above is permitted. ▮

We next consider replacing an individual tuple in the join view.

Algorithm class SPJ-R: Perform a recursive (pre-order [Knuth 73]) search on query graph tree. (We shall ignore the retracing steps that occur when leaf nodes are reached.) We are initially in State R at root relation.

STATE R (replacing): Compare projection (to this SP view) of old join view tuple with new join view tuple.

CASE R-1: Projections match exactly. Move to next relation down. Go to State R.

CASE R-2: Projections differ but keys match. Perform SP view replacement. Move to next relation down. Go to State I.

CASE R-3: Projections differ and keys differ. This can only happen in root. (For every relation but the root relation, they only ways we get back to State R—Cases R-1 and I-1—require that the keys match.) Perform SP view replacement. Move to next relation down. Go to State I.

STATE I (inserting): Compare projection (to this SP view) of old view tuple with new view tuple.

CASE I-1: Keys match. Go to State R (staying in this relation).

CASE I-2: Keys differ, new key not in SP view. Insert tuple into SP view. Move to next relation down. Go to State I.

CASE I-3: Keys differ, new projection in SP view. Move to next relation down. Go to State I.

CASE I-4: Keys differ, new key in SP view but conflicting data. Perform SP view replacement. Move to next relation down. Go to State I.

Theorem 6.4 *Algorithm class SPJ-R is the only algorithm that satisfies the five criteria that replaces a tuple from join views of the type specified when the SP views* *are identity views.*

Proof: The view tuple to be replaced can only disappear from the view if at least one of the corresponding underlying tuples have been changed. The replacement view tuple can only appear in the view if its projections appear in all of the relations appearing in the view. For join views (the SP views are identity views), the insertions and replacements specified above are performed as specified on the database. This means that algorithm class SPJ-R consists of precisely one algorithm.

The first database operation that is performed replaces a corresponding underlying tuple. This causes the original view tuple to disappear.

We need to show that the algorithm implements the view update request. We observe that as long as each database request succeeds, we do not leave a relation until it contains a projection of the view tuple. Thus, showing that the algorithm implements the view update request is reduced to showing that the each database request succeeds. We insert only when there is no tuple in the database with a matching key. We replace only when there is a tuple in the database with a matching key. Since the only constraint in each relation is a functional dependency, the insertions and replacements must succeed.

We need to show that the algorithm satisfies the five criteria. (1) The only database tuples affected are projections of the view tuple. (2) There is only one database request per relation. (3) The only relations that change are those which do not already contain projections of the view tuple. Clearly, each of these must have at least one update. (4) The replacements change precisely those attributes that change in the view. The key of a database tuple changes only in the root relation when the key of the view tuple changes. (5) There are no database deletions. ∎

6.4 Combining Joins with Selections and Projections

We need to combine select and project view algorithms with join view algorithms to get select, project, and join view algorithms. Fortunately, the natural composition works correctly.

For a given select, project, and join view, the set of view update translations (translators) is the obtained from cartesian product of the sets of the view update translations (translators) for each select and project view. We use one of the algorithms SPJ-D, SPJ-I, and SPJ-R as appropriate. Each algorithm describes how to use select and project view algorithms. This notion is captured in the following theorems.

Lemma 6.1 *Let $U\Delta1$ and $U\Delta2$ be a set of view update requests on the select and project views in sets $V\Delta1$ and $V\Delta2$ respectively, where there is at most one view update request on each view and each underlying relation is referenced in only one of the views in $V\Delta1$ or $V\Delta2$ but not both. Let $T\Delta1$ ($T\Delta2$) contain one translation for each view update request in $U\Delta1$ ($U\Delta2$). Let the sets $T\Delta1$ and $T\Delta2$ each collectively satisfy the five criteria for view update translation. Then $T = T\Delta1$´$T\Delta2$ collectively satisfies the five criteria for the view update requests $U = U\Delta1$´$U\Delta2$ on the views in $V = V\Delta1$´$V\Delta2$.*

Proof: Since the same database tuples are affected, Criterion 1 will continue to hold. Since there is at most one request per relation, Criteria 2, 4, and 5 will continue to hold.

Suppose that $T\Delta1$ and $T\Delta2$ individually satisfy Criterion 3 but T does not. Then there is a valid translation, T', for the U that is a proper subset of T. Then there is some individual update translation, say $T\Delta1$, whose translation satisfies Criterion 3 but which is not a subset of T. The only database updates in T which affect $V\Delta1$ are in $T\Delta1$. Then T`$T\Delta1$ is a proper subset of $T\Delta1$ that translates the same update. Thus, $T\Delta1$ does not satisfy Criterion 3.

A similar argument can be used to show that Criterion 3 is satisfied for the converse. ∎

Theorem 6.5 *The translation of a update on a SPJNF′ view consists of the union of a series of translations for the constituent SP views.*

Proof: The algorithm classes SPJ-D, SPJ-I, and SPJ-R each perform at most one operation per constituent SP view. Therefore, the union of the translations on the SP views satisfy all five criteria.

If there were another algorithm that satisfied the five criteria, it would have to change the SP views in the same manner as SPJ-D, SPJ-I, or SPJ-R (depending on the operation). Otherwise, there would be additional algorithms for join views. (Note that making additional changes to the SP views would have as a consequence making additional database changes, and that would violate Criterion 3.) The previous theorem shows that the translations of the updates on the SP views must use the algorithms we have described earlier. ∎

The following three theorems are a consequence of the preceding theorem and the corresponding join-only view theorems.

Theorem 6.6 *Algorithm class SPJ-D consists of the only algorithms that satisfies the five criteria that deletes a tuple from select, project, and join views of the type specified.*

Theorem 6.7 *Algorithm class SPJ-I consists of the only algorithms that satisfies the five criteria that inserts a tuple from select, project, and join views of the type specified.*

Theorem 6.8 *Algorithm class SPJ-R consists of the only algorithms that satisfies the five criteria that replaces a tuple from select, project, and join views of the type specified.*

6.5 Conclusion

We have applied the five criteria used earlier for acceptable view update translations on select and project views to select, project, and join views. We have enumerated a complete list of translations that satisfy these five criteria for a large class of select, project, and join views on Boyce-Codd Normal Form relations. Our techniques take into account the possibility that an object the user has requested to be deleted should actually be transformed into an object the user does not know about, and the possibility that an object the user wants inserted may refer to an existing object the user has becomes aware of through the very same action. Thus an object can be deleted by "destroying" it or converting it into another, unrecognizable object.

With a complete list of alternative translations, we have circumscribed the search space for a translator for view updates (into database updates). Additional semantics are needed to choose the desired translator. Collecting, coding, and using such additional semantics are discussed in the next chapter.

We handle now this large class of select, project, and join views on Boyce-Codd Normal Form relations. We require that there is a single consistency constraint on each relation in the view: a key dependency (or functional dependency). The selection condition is the (possibly empty) conjunction of terms, each of the form $attribute \in set$. The projection may remove any attributes mentioned in the selection condition, except that the key of the relation must appear in the view. The views are described in Select-Project-Join Normal Form, which requires that all the join attributes appear in the view, the joins are extension joins with inclusion dependencies, and the joins can be represented as a tree in a directed query graph.

Chapter 7

Statically Defined Views

7.1 Choosing An Update Translator

We propose that the semantics necessary for disambiguating view update translation be obtained at view definition time. The semantics are used to choose a view update translator. Once a translator is chosen, users may specify updates through the view, which the translator converts into database updates without any disambiguating dialog.

In the discussion that follows, we will assume that the view is defined by a database administrator (DBA) who will also provide the necessary semantics to choose a translator. While this is the simplest case for the use of a view definition facility, it is clear that this system could be used by any user with the wherewithal to define a view, either for the user's own use or for the use of other, perhaps less knowledgeable users. We regard the effort of collecting the semantics at view definition time to be amortized by utilizing them for many view updates.

The candidate translators can be organized into a tree, where each node of the tree represents a decision to be made. The semantics are merely the sequence of decisions made by the DBA in a walk of this tree guided by the view definition facility. The view definition facility presents questions to the DBA, each time supplying several options, based on the view definition, the database schema, and the answers to the previous questions. Note that the tree of translators is different from the query graph representing the view. Furthermore, the tree of translators is merely a pedagogical device; it does not actually exist within the view definition facility.

Choosing a translator does not use any information about the transactions that will be performed against the view. This is because the translator chosen will take as input individual view tuple updates and translate them into sets of database updates. Any information that would be contained in the nature of the transactions performed that is useful for determining how to translate the update is captured at the view definition dialog. Since the set of transactions is not necessarily available at view definition time, does not contain all the information needed for choosing a view update translator, and at best provides information that is already provided by the dialog, we have chosen to use the dialog instead. The dialog is described in subsequent sections.

7.1.1 Dialog at View Definition: Deletion

Using the query graph that defines the view, we consider the selections and projections applied to the root relation. This is because deleting from a select, project, and join view can be accomplished by deleting from the select and project view corresponding to the root relation. If there is no selection on the root relation, there is no alternative to deleting the projection of the view tuple from the underlying relation (*i.e.*, the tuple with the same key as the view tuple). If there is a selection on the root relation, we may alternatively replace the corresponding underlying tuple in the root relation by changing a selecting attribute to an excluding value.

Algorithm DBA-D: We first ask the DBA whether view tuple deletions are permitted. If there are selecting attributes in the root relation that are not part of the key, we then ask the DBA whether a deletion of a view tuple is to result in deletion of the corresponding root tuple (New York manager example) or its replacement (baseball team manager example). If the DBA chooses deletion, we know how to translate view tuple deletions. Otherwise, of the selecting attributes in the root relation that are not part of the key, we ask the DBA to choose which one of them is to be replaced (unless, of course, there is only one). In a menu-driven system, the attributes that have only one excluding value should be highlighted, since they are more likely to be desired. Once the selecting attribute to be changed is chosen, we need to decide which excluding value to use for that selecting attribute. We then ask the DBA to choose an excluding value for the chosen selecting attribute (unless there is only one). The resultant view update translation replaces the corresponding underlying tuple from the root relation by changing the chosen selecting attribute with the chosen excluding value. This view update translation does not have any side effects in the view and only affects one database tuple.

7.1.2 Dialog at View Definition: Insertion

Inserting into a selection, project, and join (SPJ) view involves ensuring that the projections of the view tuple appear in each of the relations so that they may be joined together to form the view tuple. The SPJ view can be decomposed into a join view of a series of select and project (SP) views formed by taking the selections and projections of the query graph using them to define a view on each relation. The following algorithm decomposes a SPJ view tuple insertion into a series of operations on these SP views.

Insert into SPJ view: Take the projections of the join view to the attributes listed in each SP view. On each projection (or SP view) there are three cases:

CASE 1: The projection exists in the SP view in the exact projected form. If this is the root SP view, reject the view update as it violates an FD in the view. Otherwise, we need do nothing with this SP view.

CASE 2: The projection does not match the key of any tuple in the SP view. Perform an SP view insertion using the projection of the new join view tuple.

CASE 3: There is already a tuple in the SP view with a key matching that of the projection, but the other values do not match. Replace (in the SP view) the existing SP view tuple by the projection of the new join view tuple. We may reject the update request if we do not wish to perform a replacement in the SP view.

If any of the SP view operations fail, the entire view update request fails and is undone.

Algorithm DBA-I: We first ask the DBA whether view tuple insertions are permitted. We proceed if they are permitted. At view definition time, we perform a depth-first search (in preorder) of the query graph and determine how to perform view tuple insertions. For each relation, we must determine what to do in Cases 2 and 3 above.

We then ask whether any modifications may be permitted in this relation. If not, then when updates require changes to this relation, they will be rejected. We then ask whether an insertion is permitted that will cause a new tuple to be inserted into the database. This is one of the two possibilities of Case 2, and it arises in View P above. For each attribute in this relation that does not appear in the view, we need to ask the DBA for the selecting value to use (unless there is only one). When inserting a tuple into the database, we take the projection of the view tuple to this relation and extend it with these values chosen.

Next, we ask whether inserting a view tuple can result in a change to a database tuple that does not satisfy the selection condition (as arises in the baseball team manager example). This is the other possibility of Case 2. We change the database tuple whose key matches the corresponding attributes of the view tuple so that its values match those of the view tuple. If there are selecting attributes in this relation that do not appear in the view, we will have to change every excluding value in the database tuple to a selecting value. If we have obtained a list of values according to the preceding paragraph, those will suffice here; otherwise, we ask the DBA to choose a selecting value to use for each selecting attribute in this relation that does not appear in the view (unless there is only one selecting value). The implementor of a view update facility may offer the option of allowing the DBA to indicate that the view update is to be rejected if a particular non-appearing selecting attribute does not already have a selecting value in lieu of giving a selecting value to change it to; this is only meaningful when there are multiple selecting attributes, at least one of which does not appear in the view.

Finally, we ask whether inserting a view tuple can result in a change to a database tuple that *does* satisfy the selection condition (Case 3). Such a change results in a side effect when the database tuple is one of the corresponding underlying tuples for some other view tuple. Then the change requested will affect those other view tuples that share this corresponding underlying tuple. If permitted, the translation is to change the database tuple so that it matches the projection of the view tuple to that relation.

7.1.3 Dialog at View Definition: Replacement

Replacing in a SPJ view can be decomposed into a series of replacements and insertions in select, project views on the underlying relations. The following algorithm describes this process.

Replace into SPJ view: Perform depth-first search on query graph tree. We are initially in State R at root relation.

State R (replacing): Compare projection (to this SP view) of old join view tuple with new join view tuple.

Case R-1: Projections match exactly. Move to next relation down. Go to State R.

Case R-2: Projections differ but keys match. Perform SP view replacement if allowed. Move to next relation down. Go to State I.

Case R-3: Projections differ and keys differ. This can only happen in root. Perform SP view replacement if allowed. Move to next relation down. Go to State I.

State I (inserting): Compare projection (to this SP view) of old view tuple with new view tuple.

Case I-1: Keys match. Go to State R (staying in this relation).

Case I-2: Keys differ, new key not in SP view. SP view insert tuple. Move to next relation down. Go to State I.

Case I-3: Keys differ, new projection in SP view. Move to next relation down. Go to State I.

Case I-4: Keys differ, new key in SP view but conflicting data. SP view replace if desired, else reject request. Move to next relation down. Go to State I.

Cases R-1, I-1, and I-3 require no action. Case I-2 can use the same algorithm as Case 2 from the previous section. Cases R-2 and I-4 can use the same algorithm as Case 3 from the previous section.

Case R-3 is more complicated. There are two alternative ways to remove the old database tuple based on the two alternatives for deleting a view tuple (delete and replace). There are two alternative ways to insert the new database tuple, depending on whether there is already a conflicting database tuple there. In the case when the old database tuple is to be deleted and the new one is inserted, a replacement is performed instead.

Algorithm DBA-R: We first ask the DBA whether view tuple replacements are permitted. Note that replacements may be permitted even when insertions or deletions are not permitted. When insertions or deletions are permitted, we would like replacements to be handled similarly. Thus the questions necessary for determining how to replace view tuples are the ones necessary for determining how to delete and insert view tuples. Note that if questions were omitted because the DBA decided not to permit the operation for insertion or deletion (*e.g.*, Case 2 for insertion), the DBA may be given the opportunity to decide on a more liberal answer for replacement. For example, it is reasonable to allow side effects on replacements (when the intent is clear) and not permit side effects on insertions (when the intent is not clear).

7.2 Relationship with Database Design Process

View definition may occur as part of the database design process or it may occur much later. The process of defining a view and choosing a view update translator requires some information that was available during the database design process and should be captured at that time. These involve functional and inclusion dependencies. Collection of this type of information is one of the objectives of the structural model [Wiederhold 83].

Since we require all relations to be in Boyce-Codd Normal Form, functional dependencies are key dependencies: There is a key for each relation. Commercial relational database systems have some support for keys [Astrahan 76] and could enforce the uniqueness constraint.

Inclusion dependencies are, however, not usually captured or enforced by commercial relational database systems. In earlier work, we have shown how such constraints can be enforced incrementally [Keller 81]. Note that our algorithms will work if the inclusion dependencies are not enforced with two exceptions. The first exception is that dangling join tuples—tuples that participate in an outer join but not an inner join—that do not appear in the view may cause updates to fail unexpectedly. The dangling join tuple is treated similarly to a tuple that did not appear because it did not satisfy a selection clause. The second exception is that it is no longer necessary to delete the root tuple when deleting a view tuple. Deleting the root tuple cannot have any side effects in the view, while deleting any other supporting underlying tuple may potentially cause the deletion of additional view tuples. Therefore, it is still desirable to use the "delete at the root" algorithms.

Chapter 8

Dynamically Defined Views

In this chapter, we will explore using our approach for dynamically generated views, such as universal relations [Sciore 80, Ullman 82b] or by access through natural language [Davidson 81, 83, Salveter 84a, 84b]. We shall not be interested in how the view definition is obtained, but only in that there is no expert who can choose the desired translator when given the view definition. The dynamic view update mechanism must choose the view update translator itself.

8.1 Some Differences Between Dynamic and Static Views

Static views are tailored to specific classes of users and are used for many queries and updates; dynamic views are generated on the fly based on user interaction. For a static view, it is feasible to invest time during view definition to choose a translator for the potentially numerous view updates that will be requested through that view. For a dynamic view, the number of updates through any view definition is usually much smaller. The identical view may be defined repeatedly in the same or different session, but it is unlikely that the database system will maintain a long history to correlate these views.

A user of a dynamic view has more flexibility than a user of a static view. Consider the baseball team manager from Chapter 5. To remove a player from the team, he has no choice but to request the deletion of the player from his view. With a dynamic view, the manager could explicitly request changing the player's status from a member of the team to a non-member. If the baseball team manager requests deletion of an employee, the database system can expect that the request is to delete the employee from the employee relation. For a user who acts in multiple roles (*e.g.*, if the baseball team manager also worked for personnel), these roles would correspond to separate views.

We suggest that select and project view updates be translated by using the same database operation as the view operation requested. For example, a view deletion should be translated as a database deletion rather than a database replacement. These algorithms are I-1 (for insertion), D-1 (for deletion), and R-1 and R-2 (for replacement). This will significantly reduce the ambiguities. The join view algorithm from Chapter 6 may be used in conjunction with these select and project algorithms.

8.2 Using Additional Semantics

We could attempt to capture additional semantic information global to the database. For example, based on the selection condition, we might be able to determine the nature of the view update translator desired. Let us explore some of the differences between the baseball manager's view (B) and the personnel manager's view (P) in Chapter 5.

View P:

```
Select *
From EMP
Where Location="New York"
```

View B:

```
Select *
From EMP
Where Baseball="Yes"
```

When the personnel manager requests deletion of employee #17, we could conceivably move the employee to San Francisco. But the semantics of location makes that decision a little more difficult. Rather than two locations, there are potentially numerous locations to transfer "fired" employees to.[1]

[1]That is one way to fire an employee: transfer her to a location you *know* she would not go to.

The semantics of the baseball field can be elaborated into the position of the player. The domain would be pitcher, catcher, first baseman, second baseman, ..., and "none of the above." Then View B would select *not* none of the above. Getting rid of a third baseman could mean changing his position into "none of the above" or deleting him from the employee relation altogether.

We make several observations about the differences. It is more reasonable to translate a deletion update against a view that excludes a single value into a replacement (to the excluded value) than for a view that excludes many values (or selects a single value). This is a heuristic based on syntax. Some attributes may be more mutable than others: whether someone is a member of the baseball team is perhaps of less consequence than whether someone works at the New York office. This is a semantic consideration.

There may be other examples of syntactic and semantic considerations that facilitate view update translation for dynamic views. We suggest, however, than approaches that involve heuristics choose simple view update translators that are as independent of the state of the database as possible.[2] Such considerations and approaches are generally beyond the scope of this work.

8.3 Disambiguating Dialog

There is the option of engaging in a clarifying dialog with the user when a request is ambiguous. This approach was used for queries in the Rendezvous system [Codd 78], but the users of the system found the dialog to be tedious. View updates would be yet more tedious because of the inherent ambiguities. We have avoided this option for static views by having all of the decisions made by at view definition time. There also is the problem that a disambiguating dialog may involve parts of the database with which a user is not familiar. For example, a request to change the manager of an employee results in the need to determine in which department the employee winds up. The user may not know anything about departments, but only about employees and managers.

[2]Consider the discussion of Davidson's approach in Chapter 2.

8.4 Protection and Authority Constraints

We may use protection and authorization checking to choose between the various alternative translators. In the baseball example, the baseball manager may only be allowed to change the baseball attribute. The translation of the request to delete a member of the team by deleting the employee record will fail by violating the database protection system. The translation that changes the baseball attribute, however, will succeed.

8.5 Other Constraints

Other constraints may be used to disambiguate view update translation in a manner similar to protection and authority constraints of the previous section. Note, however, that explicit functional dependencies are not amenable to such treatment, as they require changing more attributes than our algorithms specify.

8.6 Future Work on Dynamic Views

Universal relations [Ullman 82b] are a good framework for dealing with the problem of updating through dynamically defined views. If the views satisfy our requirements, we have shown how to enumerate all possible translations of a view update request. We can use this result to generate alternatives from which a heuristic approach can choose. In future work, we will consider choosing the desired translation for universal relation views.

Chapter 9

Null Values

Incomplete information is endemic to shared databases. No one user supplies the complete body of information, and it arrives in a piecemeal fashion. The problem of incomplete information is particularly clear for updating through select and project views.

When updating through project views, the attributes that do not appear in the view cannot be maintained by the user. The view update facility can supply some values by, for example, retaining existing values during a view tuple replacement. In the case of insertion of a new view tuple, the attributes omitted from the view may not have any value. Such nulls will in general represent no constraints on the values assumed.

When updating through selection views, there are two types of constraints that may arise. If the deletion of a view tuple is translated as a replacement, Algorithm D-2 specifies that an excluding value be chosen. The alternative is to use a set null [Lipski 79] that lists the excluding values. When inserting a view tuple, if a selecting attribute does not appear in the view, Algorithm I-1 specifies that a selecting value be chosen. The alternative is to use a set null that lists the selecting values.

Set nulls arise naturally in view updates, but cause problems for other views. One view may have a selection condition that depends on an attribute into which set nulls are introduced by another view. The potential for tuples which possibly satisfy the selection condition now arises. The question of how to update through views that are based on incomplete information databases is beyond the scope of this work. Updating incomplete information databases without views is in itself an important and difficult problem [Keller 84a, 85c].

Chapter 10

Limitations of the Approach

This chapter discusses the limitations of our approach. The primary limitation is on the categories of views we support. A weakness is the inability to make use of certain information that a view update facility could make use of, but ours does not. Finally, we will discuss the knowledge we do require and how necessary it is.

10.1 Class of Views

We handle a class of select, project, and join views. We have restrictions on the selections, projections, and the joins.

10.1.1 Join Restrictions

There are several restrictions on joins. The join attributes may not be removed from the view via projections. The query graph must be rooted. The query graph must be a tree. The joins must be reference connections.

The most severe restriction on joins is that join attribute may not be removed from the view via projections. This is a consequence of Select, Project, and Join Normal Form (SPJNF). We discussed the importance of this restriction in Section 1.2.1. Briefly, when the join attributes are removed from the view via projections, the view update problem becomes data dependent. Our algorithms are data independent; we decide what to do in advance without consideration of the data, although the data are used to determine whether the update will succeed or fail.

The query graph must have a root so that deletions are unambiguous. This is because deletion may only occur at the root, as a consequence of Criterion 3. This is also discussed in Section 1.2.1.

The third limitation restricts the query graph to being a tree. The algorithms can be adapted to allow rooted directed acyclic graphs (rooted DAGs), however this would imply the potential for more than one view update request on an individual SP view (the select and project expression for an individual relation). This would invalidate our proof that our translations satisfy the 5 criteria. Thus, we would need another standard for acceptable translations.

The last limitation is that the joins must be reference connections. Recall that a reference connection involves an extension joins with an inclusion dependency. The requirement for the extension join boils down to a many-to-one connection. Joins that are many-to-many suffer from the same problem as do multiple roots. Furthermore, many-to-many joins occur infrequently in database structures [Ceri 83]. The inclusion dependency is used for several reasons. It explains why Criterion 3 requires deletion at a root. It eliminates dangling tuples—tuples that participate in an outer join but not an inner join—that can cause view side effects during view insertion (the dangling tuples may suddenly appear). And it describes well the semantic integrity of the database. The algorithms we have presented can easily be adapted if inclusion dependencies are not enforced, but there may be surprising (to the user of the view) effects when the inclusion dependency is violated (i.e., when there are dangling tuples).

10.1.2 Projection Restrictions

The only restriction on projections is that the key of each relation may not be removed. This implies that join attributes may not be removed, by a requirement on joins. We illustrated at great length in Chapter 5 the unsatisfactory algorithms that we are forced to use when the keys are projected out of the view.

10.1.3 Selection Restrictions

The selection condition is a conjunction of terms of the form $A \in s$ (or equivalently, $A \notin e$), where s (and e) is a set of constants in the domain of A. We do not allow disjunctions. We do not allow comparisons between two attributes.

There are two problems that arise from disjunctions. Disjunctions between terms referring to different relations would prevent the view from being expressed in SPJNF (because joins imply the conjunction of the selection clauses for each relation). Disjunctions between terms referring to the same relation would require more complex translation algorithms. For example, when deletion of a view tuple is translated into replacement of a corresponding underlying (database) tuple, we know that at most one attribute must be changed. If the selection condition contained a disjunction, it might require changing more than one selecting attribute to accomplish the view deletion, namely the attributes that appeared in the disjunction.

There are several types of comparisons between two attributes. An equality comparison between two attributes of two different relations is a join, not a selection; it *is* allowed. An equality comparison between two attributes of the same relation would result in more complex translation algorithms.[1] These would not require more than one attribute to be changed to invalidate the selection clause (if it were a conjunction). Non-equality comparisons can involve arbitrarily complex computations to determine the values of attributes that need to be changed to satisfy the selection condition. It is possible to imbed an undecidable problem in a selection condition, although fortunately not in SQL.

10.2 Other Knowledge

There are other kinds of knowledge we could use but do not. We could use the user's view update request rather than tuple-by-tuple effects on the view. Consider the following example.

```
ED relation              DM relation
employee  department     department  manager
Joe       Art            Art         Sally
Sarah     Art            Books       Sally
George    Music          Music       Ira
Fred      Music
EDM view
```

[1]An equality comparison between two attributes of the same relation could not be a join because each relation is only allowed to appear once in the query graph.

employee	department	manager
Joe	Art	Sally
Sarah	Art	Sally
George	Music	Ira
Fred	Music	Ira

If the update request is to change the manager of everyone in the Music department from Ira to James, this will involve two replacements: ⟨`George`, `Music`, `Ira`⟩ to ⟨`George`, `Music`, `James`⟩, and ⟨`Fred`, `Music`, `Ira`⟩ to ⟨`Fred`, `Music`, `James`⟩. The latter will naturally occur as a side effect of the former; the view update facility may ask whether this side effect is desired when presented with the first replacement if it does not realize the second request is also coming. Consideration of this problem is a topic for future study.

Authorization and other structural knowledge can be useful for disambiguating view updates for dynamically defined views. This is another avenue for future study. On the other hand, there is no need to use additional knowledge that will not improve the results.

10.3 Knowledge Requirements

To determine a view update translator, we require two types of knowledge. The first is global to the database, such as keys. We expect that this information has been captured at database design time. The second is obtained in the dialog with the view definer (or DBA). We could attempt to get as much of this knowledge as possible from a knowledge base, but we have chosen to ask a series of directed questions instead. Such a knowledge base has been considered for distributed databases [Apers 84]. We leave the question of how to use a knowledge base to choose a view update translator as open for further work.

10.4 Other types of views

SPJ views provide a clear connection between the view and the underlying data. Views involving some other operators, such as aggregation, have inherent problems for updating through. Consider an employee relation that contains employee and department names. The view lists department names and number of employees for each department.

```
SELECT Dept, Number(Emp)
FROM Emp
GROUP BY Dept
```

If we decrement the number of employees for a department, the view update facility has know way of knowing which employee to delete from the employee relation. This is an example of the problems that arise when there is no key in the view or when the key in the view is not the key or foreign key of the database relations appearing in the view.

Chapter 11

Future Directions

Most of the future directions have already been mentioned earlier. Consequently, this chapter will be a brief recapitulation.

It may be useful to explore relaxing the constraints on acceptable views. Perhaps the removal of join attributes can be handled by imposing additional constraints on acceptable view updates translations. More complex selection conditions are also worthy of consideration.

The interaction of view updates and null values needs considerable study. Chapter 9 identifies some of the issues.

Handling dynamically defined views is an interesting problem. In further work, we will explore updating through a universal relation interface. In such a situation, it would probably be best to reduce the number of alternatives by translating insertions as insertions, deletions as deletions, and replacements as replacements. Authorization checking can also be used to eliminate alternatives. We propose that future work consider an algorithmic approach that does not depend on the database state which is motivated by some of the heuristics Davidson uses.

Another topic for further consideration is how additional knowledge can be used to shorten or eliminate the dialog at view definition time. Such an approach would also facilitate the use of dynamic views.

The nature of the user interface has not been directly addressed. In particular, when the view definition facility discovers that a view side effect is required, how should the user confirm that the side effect is desired?

Chapter 12

Conclusion

We have devised five criteria for acceptable view update translations. We have enumerated a complete list of translations that satisfy these five criteria for a large class of select, project, and join views on Boyce-Codd Normal Form relations. Our techniques take into account the possibility that an object the user has requested to be deleted should actually be transformed into an object the user does not know about, and the possibility that an object the user wants inserted may refer to an existing object the user has becomes aware of through the very same action. Thus an object can be deleted by "destroying" it or converting it into another, unrecognizable object.

With a complete list of alternative translations, we have circumscribed the search space for a translator for view updates (into database updates). Additional semantics are needed to choose the desired translator.

We handle this large class of select, project, and join views on Boyce-Codd Normal Form relations. We require that there is a single consistency constraint on each relation in the view: a key dependency (or functional dependency). The selection condition is the (possibly empty) conjunction of terms, each of the form *attribute* $\in$ *set of constants*. The projection may remove any attributes mentioned in the selection condition, except that the key of the relation must appear in the view. The views are described in Select-Project-Join Normal Form, which requires that all the join attributes appear in the view, the joins are extension joins with inclusion dependencies, and the joins can be represented as a tree in a directed query graph.

We have also described a sequence of questions that can choose a valid view update translator for a large class of select, project, and join views. The class of translators we choose from are based on the algorithm templates that generate all possible translations that satisfy five criteria for view update translation. The database administrator defines the view and answers the questions to choose a translator, whose definition should be stored along with the view definition. Later, users may request insertions, deletions, and replacements through the view, and these will be translated by the chosen translator into database updates without any disambiguating dialog. Side effects may result from some insertions and replacements if permitted by the database administrator; it may be desirable to have the user confirm such side effects, especially for insertions.

We do not claim that all possible translators are subject to being chosen by our questions. The set of candidate view update translators is quite large; we have bounded this set by enumerating the set of all view update translations. Some of the translators chosen here will translate all updates that have translations satisfying our criteria; others will reject some updates because they were proscribed by the answers by the database administrator (DBA) to the questions asked by the view definition facility. These translators are simple; in fact, they were used to generate the enumeration of translations.

The process of defining a view and choosing a translator has been described here as being performed by the DBA. While this is the simplest case for the use of such a system, it is clear that this system could be used by any user with the wherewithal to define a view. The distinction to make is that such a dialog would be most effective for static views that are defined once and used repeatedly. For dynamic views, defined by natural language dialog or universal relation interfaces, the overhead of answering the questions would not be amortized over performing many view updates. Heuristics and user profiles could be used to determine the answers we need to choose a translator [Davidson 81].

Querying and updating through a view reduces the security and protection problem, but does not eliminate it. Clearly, a view circumscribes the collection of data a user is permitted to access. The question of how to give each manager access to the data for that department can be addressed either by a parameterizing the view to only show that department's data or by parameterized protection scheme that allows access only to tuples containing data for that department. Using both may seem redundant but need not be. A parameterized view will make fewer demands on the database and the security system. A security system could have a large loophole if it gave special consideration to queries and updates specified through views. Of course, an effective security system is needed when a view definition facility and an *ad hoc* query facility is made available to users.

With views and queries described non-procedurally, relational databases are an effective tool for productivity [Codd 82]. We have shown how to describe view update translators non-procedurally by answering a sequence of questions based on the view definition and the database structure. This has the potential to dramatically increase the productivity of views, and consequently, relational databases.

We have considered single query views in this work. This results in a single relation. For multiple relations presented to the user, multiple view queries can be defined. We call the mixture of individual view queries and database relations presented to the user a *window*. We leave consideration of the differences between queries against windows for future work.

References

[Apers 84] P.M.G. Apers and Gio Wiederhold, "How to Survive a Network Partition," submitted for publication, April 1984.

[Astrahan 76] M. M. Astrahan, *et al.*, "System R: Relational Approach to Database Management," *Trans. on Database Systems*, **1**, 2, ACM, June 1976.

[Bancilhon 79] F. Bancilhon, "Supporting View Updates in Relational Data Bases," in *Data Base Architecture*, Bracchi and Nijssen, eds., North Holland, June 1979.

[Bancilhon 81] F. Bancilhon and N. Spyratos, "Update Semantics of Relational Views," in *Trans. on Database Systems*, ACM, **6**, 4, December 1981.

[Birkhoff 67] Garrett Birkhoff, *Lattice Theory*, American Mathematical Society, Providence, RI, Colloquium Publications, Volume 25, 1967.

[Carlson 79] C. Robert Carlson and Adarsh K. Arora, "The Updatability of Relational Views Based on Functional Dependencies," *Third International Computer Software and Applications Conference*, IEEE Computer Society, Chicago, IL, November 1979.

[Casanova 82] Marco Casanova, Ronald Fagin, and Christos Papadimitriou, "Inclusion Dependencies and Their Interaction with Functional Dependencies," *Proc. of the ACM Symp. on Princ. of Database Systems*, Los Angeles, March 1982.

[Ceri 83] S. Ceri, S. Navathe, and G. Wiederhold, "Distribution Design of Logical Database Schema," in *Trans. on Software Eng.*, **SE-9**:4, IEEE, July 1983.

[Chamberlin 75] D. D. Chamberlin, J. N. Gray, I. L. Traiger, "Views, Authorization, and Locking, in a Relational Data Base System," *Proc. National Computer Conference*, AFIPS, 1975.

[Codd 78] E.F. Codd, "How About Recently? (English Dialogue with Relational

Databases Using RENDEZVOUS Version 1)," in *Databases*, Shneidermann (ed.), Academic Press, NY, August 1978.

[Codd 82] E.F. Codd, "Relational Database: A Practical Foundation for Productivity," *Comm. Assoc. Comput. Mach.*, **25**:2, Feb. 1982. The 1981 ACM Turing Award Lecture, delivered at ACM, Los Angeles CA, Nov. 9, 1981.

[Cosmadakis 83] Stavros S. Cosmadakis and Christos H. Papadimitriou, "Updates of Relational Views," *Proc. of the ACM Symp. on Princ. of Database Systems*, (Atlanta, Georgia), March 1983.

[Cosmadakis 84] Stavros S. Cosmadakis and Christos H. Papadimitriou, "Updates of Relational Views," in *Journal of the Assoc. Comput. Mach.*, **31**:4, October 1984.

[Davidson 81] Jim Davidson and S. Jerrold Kaplan, "Natural Language Access to Databases: Interpretation of Update Requests," *Proc. 7th Int. Joint Conf. on Artificial Intelligence*, Vancouver, B.C., August 1981.

[Davidson 83] James E. Davidson, "Interpreting Natural Language Database Updates," Stanford University, Computer Science Dept., Ph.D. dissertation, December 1983.

[Dayal 78] U. Dayal and P. A. Bernstein, "On the Updatability of Relational Views," *Proc. Fourth VLDB Conf.*, IEEE Computer Society, Berlin, West Germany, October 1978.

[Dayal 79] Umeshwar Dayal, *Schema-Mapping Problems in Database systems*, Aiken Computation Laboratory, Harvard University, TR-11-79, Ph.D. dissertation, August 1979.

[Dayal 82] U. Dayal and P. A. Bernstein, "On the Correct Translation of Update Operations on Relational Views," *ACM Trans. on Database Systems*, **7**:3, September 1982.

[El-Masri 79] Ramez El-Masri and Gio Wiederhold, "Data Models Integration using the Structural Model," *Proc. of the 1979 SIGMOD Conference*, ACM SIGMOD, Boston, June 1979.

[El-Masri 80] Ramez El-Masri, *On the Design, Use, and Integration of Data Models*, Ph.D. dissertation, Stanford University, 1980.

[Finkelstein 82] Sheldon Finkelstein, "Common Expression Analysis in Database Applications," *Proc. Int. Conf. on Management of Data*, ACM SIGMOD, Orlando, FL, June 1982.

[Furtado 79] A. L. Furtado, K. C. Sevcik, and C. S. Dos Santos, "Permitting Updates Through Views of Data Bases," *Inform. Systems*, **4**:4, Pergamon Press, Great Britain, 1979.

[Halmos 60] Paul R. Halmos, *Naive Set Theory*, Springer-Verlag, New York, 1960.

[Honeyman 80] Peter Honeyman, "Extension Joins," *Proc. Int. Conf. on Very Large Data Bases*, Montreal, 1980.

[Keller 81] Arthur Keller and Gio Wiederhold, "Validation of Updates Against the Structural Database Model," *Proc. Symposium Reliability in Distributed Software and Database Systems*, IEEE Computer Society, Pittsburgh, PA, July 1981.

[Keller 82] Arthur M. Keller, "Updates to Relational Databases Through Views Involving Joins," in *Improving Database Usability and Responsiveness*, Peter Scheuermann, ed., Academic Press, New York, 1982.

[Keller 84a] Arthur Keller and Marianne Winslett Wilkins, "Approaches for Updating Databases With Incomplete Information and Nulls," in *IEEE COMPDEC Computer Data Engineering Conference*, (Los Angeles), April 1984.

[Keller 84b] Arthur M. Keller and Jeffrey D. Ullman, "On Complementary and Independent Mappings on Databases," *1984 ACM SIGMOD Int. Conf. on Management of Data*, Boston, June 1984.

[Keller 85a] Arthur M. Keller, "Choosing a View Update Translator by Dialog at View Definition Time," submitted for publication.

[Keller 85b] Arthur M. Keller, "A Reasonable View Update Translator That Preserves No Complement," submitted for publication.

[Keller 85c] Arthur Keller and Marianne Winslett Wilkins, "On the Use of an Extended Relational Model to Handle Changing Incomplete Information," to appear in *Trans. on Software Eng.*, IEEE.

[Knuth 73] Donald E. Knuth, *The Art of Computer Programming, Volume 1, Fundamental Algorithms*, Addison-Wesley, Reading, MA, second edition, 1973.

[Lipski 79] Witold Lipski, Jr., "On Semantic Issues Connected with Incomplete Information Databases," in *ACM Trans. on Database Systems*, **4**:3, September 1979.

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

[Masunaga 83] Y. Masunaga, "A Relational Database View Update Translation Mechanism," IBM, San Jose Reserach Laboratory, Report RJ3742, 1983.

[Rissanen 77] Jorma Rissanen, "Independent Components of Relations," *ACM Trans. on Database Systems*, **2**:4, December 1977.

[Salveter 84a] Sharon Salveter, "A Transportable Natural Language Database Update System," *Proc. of the ACM Symp. on Princ. of Database Systems*, (Waterloo, Ontario, Canada), April 1984.

[Salveter 84b] Sharon Salveter, "Supporting Natural Language Database Update by Modeling Real World Actions," in *Proc. of the First Int. Workshop on Expert Database Systems*, Larry Kirschberg (ed.), (Kiawah Island, SC), October 1984, Institute of Information Management, Technology and Policy, College of Business Administration, University of South Carolina, Columbia, SC 29208.

[Sciore 80] Edward Sciore, *The Universal Instance Assumption and Database Design*, Ph.D. dissertation, Princeton University, October 1980.

[Stonebraker 75] Michael Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," *Proc. of the 1975 SIGMOD Conference*, ACM SIGMOD, San Jose, June 1975.

[Ullman 82a] Jeffrey D. Ullman, *Principles of Database Systems*, Computer Science Press, Potomac, MD, second edition, 1982.

[Ullman 82b] Jeffrey D. Ullman, "The U.R. Strikes Back," in *Proc. of the ACM Symp. on Princ. of Database Systems*, Los Angeles, March 1982.

[Wiederhold 83] Gio Wiederhold, *Database Design*, McGraw-Hill, Second edition, 1983.