MAINTAINING DATA WAREHOUSES UNDER LIMITED SOURCE ACCESS

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
Nam (Pierre) Huyn
August 1997

Abstract

A data warehouse stores views derived from data that may not reside at the warehouse. Using these materialized views, user queries can be answered quickly because querying the external sources where the base data reside is avoided. However, when the sources change, the views in the warehouse can become inconsistent with the base data and must be maintained. A variety of approaches have been proposed for maintaining these views incrementally. At the one end of the spectrum, the required view updates are computed without restricting which base relations can be used. View maintenance with this approach is simple but can be expensive, since it may involve querying the external data sources. At the other end of the spectrum, additional views are stored at the warehouse to make sure that there is enough information to maintain the views without ever having to query the data sources. While this approach saves on external source access, it may require a large amount of information to be stored and maintained at the warehouse. In this thesis, we propose an intermediate approach to warehouse maintenance based on what we call *Runtime View Self-Maintenance*, where the views are incrementally maintained without using all the base relations but without requiring additional views to facilitate maintenance. Under limited information, however, maintaining a view unambiguously may not always be possible. Thus, the main questions in runtime view self-maintenance are:

- *View self-maintainability.* Under what conditions (on the given information) can a view be maintained unambiguously with respect to a given update?
- *View self-maintenance.* If a view can be maintained unambiguously, how do we maintain it using only the given information?

The information we consider using for maintaining a view includes:

- At least the contents of the view itself and the update instance
- Optionally, the contents of other views in the warehouse, functional dependencies the base relations are known to satisfy, a subset of the base relations, and partial contents of a base relation.

Developing efficient complete solutions for the runtime self-maintenance of conjunctive-query views is the main focus and the main contribution of this thesis.

Acknowledgements

First, I would like to express my gratitude to my thesis advisor Jeff Ullman. Jeff has provided me support, encouragement, and invaluable advice throughout my Ph.D. program, and most importantly, has spent a great deal of his time going over my writing and generating sharp criticisms. I would also like to thank Serge Abiteboul and Mike Genesereth who agreed to read my thesis and helped improve it considerably, and Gio Wiederhold and Ted Shortliffe who served on my oral committee. Finally, I would like to extend my thanks to Jennifer Widom for her advice on data warehousing research, and to Gio Wiederhold for his advice on my future career paths.

I appreciate everyone in my family for their love, patience, and support, without which this thesis would not have become a reality. Special thanks go to my son Joojay, who has kept me in a playful mood when I am not working on my research, to my wife Hua-Ching who has given me unconditional support during all these years, and to my mother Tuen-Ying for her continued encouragement. Finally, I would like to dedicate this thesis to my father Yiu-Wah, who I am sure would have been very happy to witness the completion of my doctoral program had he been still with us.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Data Warehousing Environments

In the past three years, TMart, a large retail chain, has been using a data warehouse to consolidate vast amounts of information pertaining to the retailer's business such as sales, customers, suppliers, and inventories. For TMart, this information has many uses that range from mundane report generation to sophisticated marketing analyses that help the company design sales promotion, decide on which popular merchandises to carry, discover new market segments, and make long term plans for expansion. As illustrated in Figure 1.1, the raw data that the warehouse depends on comes from a variety of sources, including the point-of-sales databases used in the local TMart stores. Currently, the source data is collected, processed, and loaded into the warehouse on a weekly basis. But plans are already in place to have the warehouse refreshed more frequently, because TMart believes the information the warehouse provides is valuable and has contributed to help the company gain a competitive advantage in the retail business.

In a typical data warehouse architecture, the data that is subject to analysis is decoupled from the data produced at the sources. This decoupling provides the following benefits:

- Information can be organized in a form that makes it easy to use for applications. The transformation of raw source data, called *base data*, into highly organized information, called *views*, can range from simple data replication to arbitrarily complex processing.
- Information is available independently of the availability of the sources, since it is stored at the warehouse. The views are said to be *materialized.*
- Information can be structured and stored so as to optimize processing of queries against the warehouse. Both information availability and query performance are important because the queries generated by the applications performing analysis are complex and typically take hours and even days to complete.
- Only minimal cooperation is required from the sources to keep the warehouse in sync when the sources change. The warehouse has the burden to keep itself up to date,

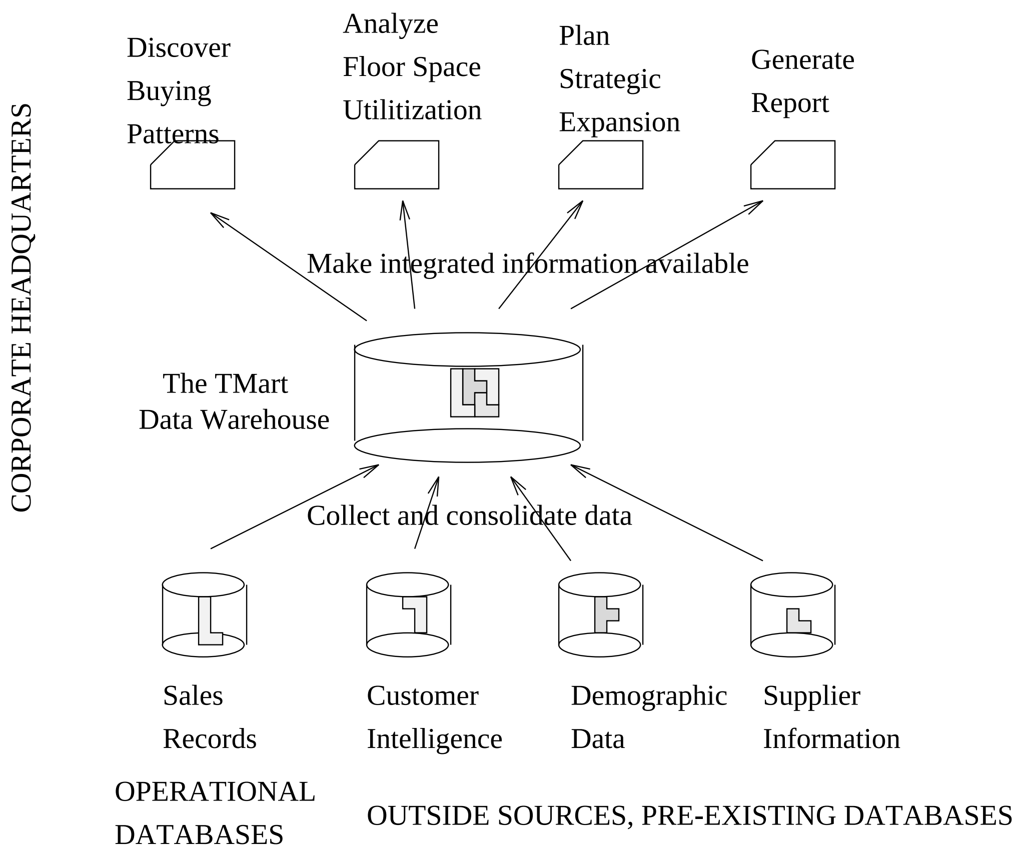


Figure 1.1: TMart's data warehouse.

and the sources merely notify the warehouse of their changes. Minimal involvement of the sources in maintaining the views in the warehouse is critical for many reasons. These sources can be operational databases engaged in recording high-volume business activities. Imposing additional load on them is not desirable. More importantly, data can also be fed from outside sources over which we have little control.

To appreciate the data warehouse architecture, it is useful to contrast it with traditional architectures and to see why these architectures are no longer sufficient to meet the demands imposed by modern information environments. Let us take a brief look at two of these traditional architectures: the query mediation architecture and the monitor architecture.

In the *query mediation architecture*, views that derive from base data are provided for answering user queries. However, these views are virtual ([Sto75]), i.e., they are not materialized as in the case of a warehouse. A user query must be decomposed into subqueries that are executed by the data sources. An example system that follows this research paradigm is the Stanford TSIMMIS system ([Pap96, GM*95]) for information integration. The query mediation architecture is depicted in Figure 1.2. As a positive consequence, answers to user queries are always based on current data. But as a negative consequence, query performance may be severely degraded, especially if the data needed to answer the user query are scattered across several sites. In the extreme case where a source holding the needed data is not accessible, the query just cannot be completely processed.

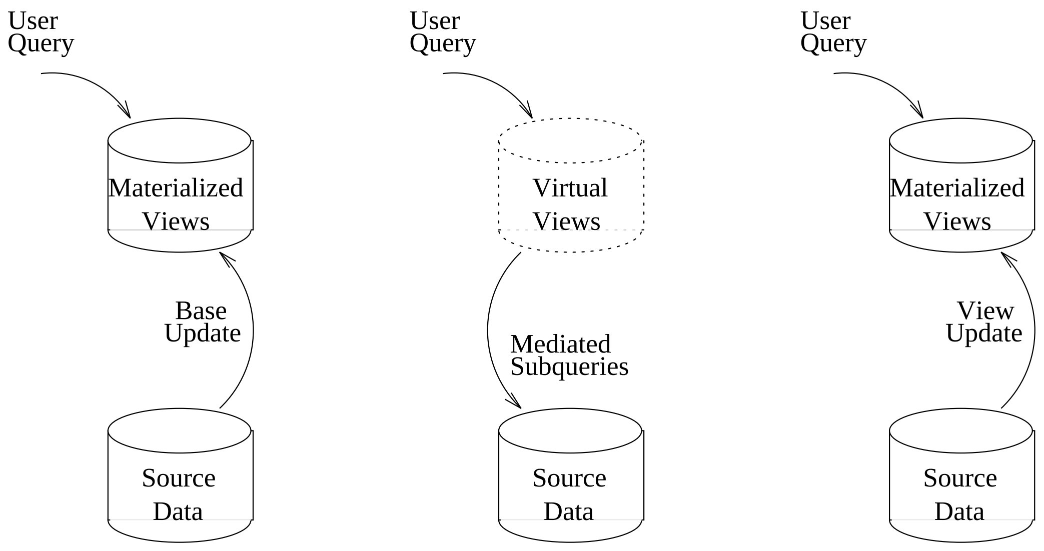
In the *monitor architecture*, views are materialized just like the case of a warehouse. As the base data changes over time, these views can also become out of sync with the base data. Thus, the need for view maintenance is common to both the monitor architecture and the warehouse architecture. But their main difference lies in where view maintenance is performed. In the monitor architecture, this responsibility rests with the sources, where additional software modules called *monitors* are installed to detect base data changes and to determine how the views are affected. Example systems that follow this research paradigm include an extension of the IBM Starburst system to maintain materialized views ([CW91, WCL91]) and a view monitoring service in the ConceptBase system ([SJ96]) that is used to refresh views materialized at a lightweight client. As a positive consequence, the sites that manage the views have little work to do to refresh the views. But the negative consequence is that additional work load is imposed on the data sources to maintain the externally materialized views, which is not desirable as mentioned before.

While data warehouses are conceived to better meet the needs in modern information environments and to deliver higher query performance than the traditional architectures can, they also create new challenges we must face now: how can we efficiently maintain the views at the warehouse? Note that although the view maintenance problem has been considered in the past, traditional work assumes a very different cost profile. In these work, base data access is no more expensive than view access. In the data warehouse architecture by contrast, we assume that accessing the base data in the sources is more expensive than accessing the views in a warehouse.

Let us now explain the major approaches to view maintenance in data warehouses.

Full Recomputation: The warehouse is taken down periodically (typically nightly or even weekly) for scheduled maintenance. During maintenance downtime, all the views are rederived from scratch from the data sources. This approach is the used by most of today's data warehouses. It is simple to implement, and changes made to the base data during the operation hours require no processing. However, this approach assumes that it is acceptable to use stale data (say yesterday's data). Further, recomputing the views from scratch when only a small fraction of the base data changes is potentially wasteful. Finally, the approach assumes a scheduled downtime long enough to complete the maintenance process. This assumption may no longer be valid when users (say across the world) expect the warehouse to be in full operation all the time or when some data source becomes inaccessible for an extended period of time.

Incremental Maintenance: Instead of recomputing every view from scratch, only the parts of the warehouse that change are computed. This approach has one of two flavors. First, maintenance can be deferred. The warehouse is scheduled for periodic



Data warehouse architecture where queries are served efficiently but warehouse is responsible for view refresh.

Query mediation architecture where queries are answered based on current data but with degraded performance.

Monitor architecture where queries are served efficiently but sources are responsible for view refresh.

Figure 1.2: Data warehouse vs. traditional architectures.

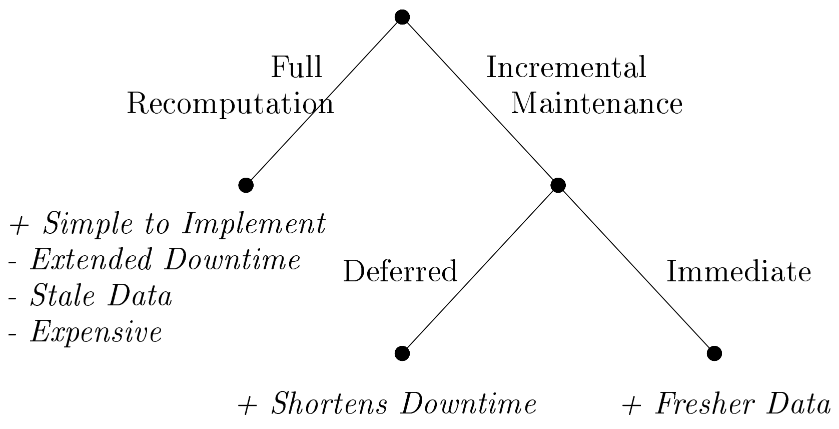


Figure 1.3: The major approaches for maintaining data warehouses.

maintenance like before, but the views are incrementally maintained rather than rederived from scratch. This method requires all changes made to the data sources during the operation hours to be logged. Compared to the full recomputation approach, this method can reduce drastically the time required to complete the maintenance process. Second, maintenance can be dynamic. The views are updated to reflect the changes made to the sources as soon as they are reported. This method has the advantage that the views provided by the warehouse are based on fresh data.

These approaches to view maintenance in data warehouses are summarized in Figure 1.3. While the approach based on full recomputation is adequate in current data warehousing environments, it may no longer meet the need for higher warehouse update performance in future environments. According to [RedBrick White Paper 96], there is a trend toward demand for 24x7 availability and for more frequent refresh (say daily or even hourly). Also, with data warehouses rapidly growing in size ([Information Week 9/96] estimates more than a few dozens terabyte-sized warehouses currently deployed, and this number is growing fast), a warehouse maintenance approach based on full recomputation just does not scale.

In this thesis, we will focus on the approach to warehouse maintenance based on incremental view maintenance. Except for the very simple views that derive from single base relations, determining the incremental changes to views that combine more than one base relation can be expensive, since the computation may involve looking up the base relations. As mentioned before, base access from the data warehouse can be expensive. Thus, the problem of incremental view maintenance in the data warehouse context is how to make the maintenance process efficient.

1.2 Approaches to Incremental View Maintenance

There are essentially three approaches to incremental view maintenance for data warehouses:

- Unrestricted Base Access.
- Self-Maintainable Warehouses.
- Runtime Warehouse Self-Maintenance.

The third approach, *Runtime Warehouse Self-Maintenance*, is the focus of this thesis. This section provides the motivation and a better understanding of our approach by way of contrasting it with the other two prevailing approaches to incremental view maintenance for data warehouses. We will close this section with a sketch of how the three approaches should play together in an overall strategy for efficient warehouse maintenance.

1.2.1 Unrestricted Base Access

This approach essentially places no restriction on the use of base relations, when a view is incrementally maintained in response to an update to some base relation.

EXAMPLE 1.2.1 Consider a data warehouse that consists of only view V, as shown in Figure 1.4. View V is defined to be the join [1] $R \bowtie S \bowtie T$ of base relations R, S, and T. Suppose a set of tuples δR is inserted into base relation R. To maintain V, we compute the increment to V by taking the join $\delta R \bowtie S \bowtie T$, and we insert the result into V. $\square$

The advantages of this approach are:

- View maintenance is conceptually simple.
- Efficient methods for computing the view increment have been well studied in the literature [Kuc91, GMS93, GM95, SJ96] for a large class of view definitions.

The approach has the following drawbacks:

- To maintain V in response to an insertion to R requires accessing base relations S and T. These external accesses can be expensive.
- Despite its simplicity, a naive way of applying this approach to maintain views for data warehouses may lead to erroneous results. This problem is referred as the *View Update Anomaly Problem.* For instance, relations S and T in the example above can be accessed in a state that is different from the state they were in when the change was made to R, because of intervening changes made to S and T. In some cases, incorrect updates to V might result.

This approach to warehouse maintenance is being actively pursued by [Z*95, ZWG97] in which methods for dealing with the view update anomaly problem are proposed.

[1]We assume the reader is familiar with the relational algebra notation. For more details, we refer the reader to [Ull89]. But briefly, a join is an operator that essentially looks for combinations of tuples with matching values in designated attributes.

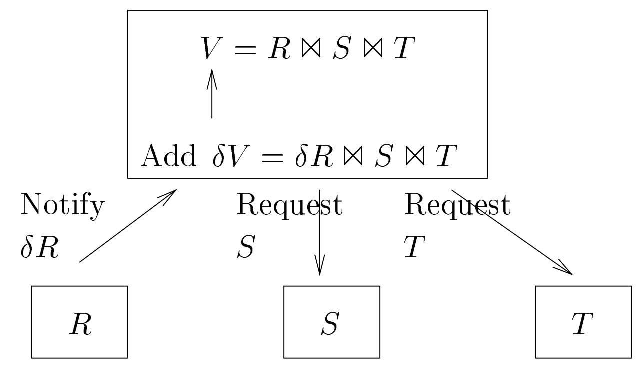

$V = R \bowtie S \bowtie T$
Add $\delta V = \delta R \bowtie S \bowtie T$
Notify δR
Request S
Request T
R
S
T

Figure 1.4: View maintenance with unrestricted source access.

1.2.2 Self-Maintainable Warehouses

The basic idea here is to maintain the views in the warehouse *without using any base relations*. With this approach, expensive external access can be eliminated altogether, and the view update anomaly problem avoided. However, as a result of not using all the base relations, there may be situations where there is not enough information to maintain a view unambiguously. Consider for instance the scenario from Example 1.2.1. Clearly, there is no maintenance expression that is a function of only V and δR and that can always maintain V correctly. Such situations never arise in traditional work on materialized view maintenance [Kuc91, GMS93, GM95, SJ96] where all the base data is assumed to be available.

Let us now clarify the notion of view self-maintainability upon which this entire approach is based.

Definition 1.2.1 (**Compile-Time Self-Maintainability**) A collection of views is said to be *(compile-time) self-maintainable* under a class of base updates if the views can *always* be maintained using only the views themselves and the base update. Note that self-maintainability is guaranteed *independently* of the actual contents of the view instance and of the base update instance. □

This notion of compile-time self-maintainability was pioneered in [TB88, BCL89, GJM96]. The gist of the self-maintainable warehouse approach is this: if the given views in a warehouse are not self-maintainable (under a class of updates), we can always materialize auxiliary views that make the final view collection self-maintainable.

EXAMPLE 1.2.2 Consider the same data warehouse as in Example 1.2.1 and consider insertions of δR to R. As mentioned before, using only V and δR, we cannot always maintain V in response to δR. But suppose in addition to V, we also materialize $V_{aux} = S \bowtie T$. Then, we can always propagate δR to view V by simply inserting $\delta V = \delta R \bowtie V_{aux}$, as illustrated

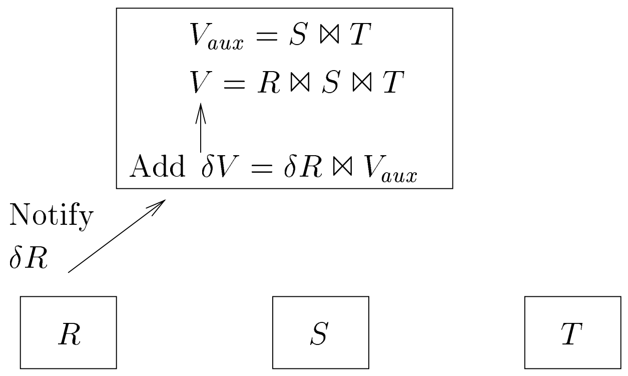

$V_{aux} = S \bowtie T$
$V = R \bowtie S \bowtie T$
Add $\delta V = \delta R \bowtie V_{aux}$
Notify δR
R
S
T

Figure 1.5: Making a warehouse self-maintainable.

in Figure 1.5. View V_{aux} itself is not affected by δR. Thus, the data warehouse, augmented with the auxiliary view V_{aux}, is now self-maintainable under insertions to R. Note that we have assumed S and T do not change. If we allow any of R, S, or T to change, we would need to materialize all the base relations for the warehouse to be self-maintainable. □

Materializing all the base relations always makes the warehouse self-maintainable, but can be wasteful. Thus, the main issue in this approach is to minimize the amount of auxiliary views to materialize at the warehouse to facilitate self-maintenance.

The self-maintainable warehouse approach has the main advantage of providing for efficient maintenance, since:

- Base access is totally eliminated.
- At runtime, there is no need to determine whether or not a view is self-maintainable, since all the views are self-maintainable by design.

The drawbacks of this approach are:

- A given collection of views that *facilitates query evaluation* has no reason to be self-maintainable. Auxiliary views that facilitate maintenance must be added, at the expense of extra storage and maintenance costs.
- In the absence of additional information on the base relations (e.g. integrity constraints), if we allow changes to be made to any base relation, then making the warehouse self-maintainable could amount to materializing every base relation. In general, the more kinds of updates to the base data we allow, the more views need to be materialized.

Thus, the key research issue in the self-maintainable warehouse approach is how to minimize the auxiliary views that must be materialized, with some appropriately defined notion of minimality. The design problem must take advantage of all the views that are already materialized, the types of base updates that are allowed, and the constraints satisfied by the base relations. For example, based on the views that are materialized to satisfy queries from users of the warehouse, the fact that some base relations never change or are append only, and the knowledge of the data dependencies that hold in the base data, what is a minimal auxiliary materialization needed in order to make the resulting warehouse self-maintainable? The self-maintainable warehouse approach is taken by [Q*96, Qua97], which studied the optimal design problem only with the knowledge of key constraints and inclusion dependencies.

1.2.3 Runtime Warehouse Self-Maintenance

The compile-time notion of self-maintainability used in the previous approach is often too conservative. There, the guarantee that a view be self-maintainable under some update class is provided for *every* possible instance of the view and of the base update. Yet, the use of such a strong guarantee cannot be avoided in the self-maintainable warehouse approach, since we do not know the exact contents of the view and of the update at design time.

The approach we are about to describe, which we take in this thesis, is based on a weaker notion of self-maintainability, called *runtime self-maintainability*, in which the self-maintainability guarantee is provided only for a specific instance of the view and of the base update.

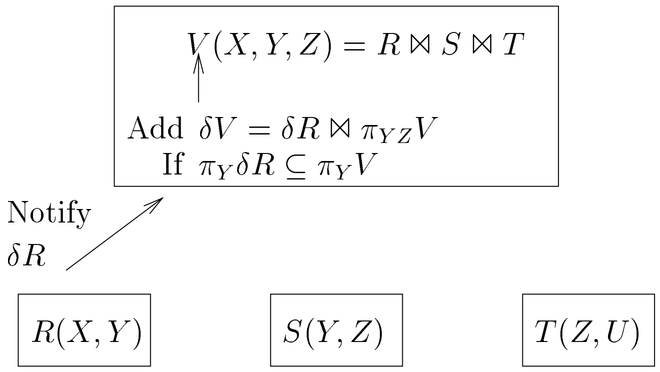
Definition 1.2.2 (**Runtime Self-Maintainability**) Consider a collection of views to maintain. An instance of the views is said to be *(runtime) self-maintainable* under a base update instance if the view instance can be maintained using only the views themselves and the base update. Note that self-maintainability is guaranteed only for a specific instance of the views and the base update. $\square$

EXAMPLE 1.2.3 Consider the same data warehouse as in Example 1.2.1 and consider insertions of δR to R. In order to explain how runtime self-maintenance works, we need to be more specific about the relation schemas used. So suppose we have the following schemas: $V(X,Y,Z)$, $R(X,Y)$, $S(X,Z)$, and $T(Y,Z)$. Even though we cannot guarantee self-maintainability of V under δR for all instances of V and δR, we can still guarantee it for certain instances of V and δR, namely those instances such that $\pi_Y \delta R$ [2] is contained in $\pi_Y V$. To maintain view V, insert $\delta V = \delta R \bowtie \pi_{YZ} V$. Figure 1.6 illustrates this approach. $\square$

Thus, this approach is the more aggressive one since it may succeed in maintaining a view where an approach based on the compile-time notion of self-maintainability may fail.

The notion of runtime view self-maintainability originates with [TB88, GB95]. However, without extensions, using the original notion (as defined in [TB88, GB95]) in the runtime warehouse self-maintenance approach would limit the practicality of the approach. For

[2] π is an operator in relational algebra that retains only the values in the specified attributes.



$V(X,Y,Z) = R \bowtie S \bowtie T$
Add $\delta V = \delta R \bowtie \pi_{YZ} V$
If $\pi_Y \delta R \subseteq \pi_Y V$
Notify δR
$R(X,Y)$
$S(Y,Z)$
$T(Z,U)$

Figure 1.6: Runtime warehouse self-maintenance.

instance, when we determine that a view cannot be maintained using only the views in the warehouse, we must consider using some of the base relations (but not necessarily all of them). Only when we fail at maintaining the view using only subsets of the base relations do we resort to using all the base relations. Thus, part of our contribution in this thesis is to generalize the notion of runtime view self-maintainability.

The advantages of the runtime warehouse self-maintenance approach are as follows:

- The approach deals with the views in the warehouse as given, without requiring auxiliary views to be materialized.
- The approach totally avoids base access, or minimizes it otherwise.

The approach has its own drawbacks:

- We must decide self-maintainability at runtime. Such tests were not needed in the previous approaches.
- Self-maintainability tests may be complex.

1.2.4 How Runtime VSM complements the other approaches

The three approaches to incremental warehouse maintenance we just described do not exclude each other. In fact, a comprehensive package for efficient warehouse maintenance should includes all three approaches.

Assume for a moment we would like to design a self-maintainable warehouse. But because of cost constraints, we may end up with a warehouse that is only partially self-maintainable: some views remain non-self-maintainable (in the compile-time sense), and materializing additional views is not an option. But how do we maintain these non-self-maintainable views? One option is to use the first approach, that is, to maintain them using

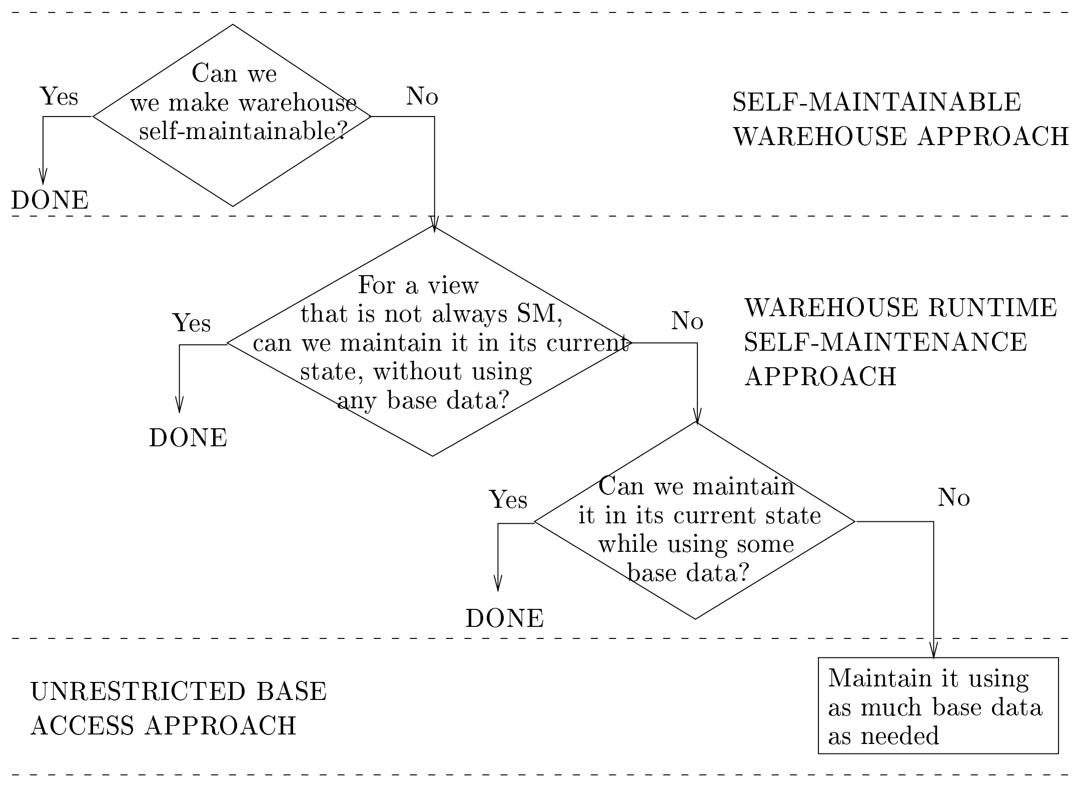


Figure 1.7: A comprehensive package for efficient warehouse maintenance.

all the base relations. While this option is certainly valid, it may be wasteful, especially if the views in question are in a state that makes them self-maintainable in the runtime sense. Thus, the runtime warehouse self-maintenance approach can be used to fill the gap between the other two approaches, as illustrated in Figure 1.7.

1.3 The Problem of Runtime View Self-Maintenance

In this section, we formally define the problem of runtime view self-maintenance (abbreviated *VSM* hereafter). We first lay out the fundamental questions in runtime view self-maintenance. We then describe each of the three dimensions of the problem space.

A warehouse is modeled as a collection of views. Each view is defined by a query over some database D. These view definitions are available to the warehouse. Other pieces of information may also be available to the warehouse, such as integrity constraints that database D satisfies.

Initially, the views are assumed to be consistent with database D. When database D

is modified, the base update U is sent to the warehouse. Since the views may become inconsistent with the new database $U(D)$, the main task of the warehouse manager is to update the views so that they become consistent with the new database.

1.3.1 The Main Questions in View Runtime Maintenance

To maintain a view V incrementally, the information we are given includes at least the following:

- A query Q that defines the view V.
- The instance V of the view itself.
- The update instance U.

Other information, denoted $\mathcal{I}$, may also be given.

View Self-Maintainability

In traditional incremental view maintenance settings, the full contents of database D is assumed to be available. Since this assumption no longer holds in our problem, a new question that was never raised before must be addressed now: whether or not we have sufficient information to bring a view up to date unambiguously. We call this the *view self-maintainability* question.

Definition 1.3.1 (**Self-Maintainability**) Given Q, V, U and $\mathcal{I}$, view V is said to be *self-maintainable* under U if $Q(U(D))$ does not depend on D, provided that D is consistent with V and $\mathcal{I}$. More formally:

$$(\forall D_1, D_2)\ Q(D_1) = Q(D_2) = V \wedge [D_1 \text{ and } D_2 \text{ consistent with } \mathcal{I}] \Rightarrow Q(U(D_1)) = Q(U(D_2))$$

$\square$

This definition is depicted in Figure 1.8.

The following terminology will be used throughout the thesis: we say that two databases D_1 and D_2 *derive differently* after the base update U when $Q(U(D_1)) \neq Q(U(D_2))$. Alternatively, we say that D_1 derive differently from D_2 after update. Note that in the phrase "derive differently", the query through which the databases derive is understood to be the query that defines the view we would like to maintain (Q in this case). When we use the phrase, it should be clear from context which view we are trying to maintain and consequently which query we have in mind.

Thus, in runtime view self-maintenance, view self-maintainability is a function of not only the view definition Q, but also the view instance V, the update instance U, and any additional information $\mathcal{I}$.

Self-maintainability of a view has the following significance:

$$\begin{tikzcd}
 & \mathcal{I} \arrow[dl, dash, "\text{consistent with}"'] \arrow[dr, dash, "\text{consistent with}"] & \\
D_1 \arrow[r, "Q"] \arrow[d, "U"'] & V & D_2 \arrow[l, "Q"'] \arrow[d, "U"] \\
U(D_1) \arrow[r, "Q"] & \overset{?}{=} & U(D_2) \arrow[l, "Q"']
\end{tikzcd}$$

Figure 1.8: View self-maintainability.

- Given information $\mathcal{I}$, if a view V is self-maintainable under update U, we are guaranteed that there is a unique new state for the view that is consistent with the updated database. However, the answer does not directly tell us how to compute the new state for the view.
- If a view V is not self-maintainable under update U, we are no longer guaranteed that a unique new state exists, *to the extent of our knowledge*. Note that the previous statement does not mean that a unique state for the view does not exist. It merely states that there are different cases with a different new state for the view, and that there is not sufficient information in $\mathcal{I}$ to make the distinction.

View Maintenance

Only when a view V is self-maintainable under an update U does it make sense to ask the question of how to bring view V up to date.

Definition 1.3.2 (**Maintenance Expression**) Given Q, V, U and $\mathcal{I}$, assume V is self-maintainable under U. A *maintenance expression* M is a program that makes view V consistent with the modified database. In other words:

$$(\forall D)\ Q(D) = V \wedge [D \text{ consistent with } \mathcal{I}] \Rightarrow M(V) = Q(U(D))$$

□

Note that the maintenance expression we need to find is a function of Q, V, U and $\mathcal{I}$.

View Independent of Updates

A special case of view self-maintainability is when a view, consistent with some database prior to an update, remains consistent with the database after the update. In other words, the base update does not affect the view, and the view does not require any update.

Definition 1.3.3 (**View Independent of Update**) Given Q, V, U and $\mathcal{I}$, view V is said to be *independent of update* U if:

$$(\forall D)\ Q(D) = V \wedge [D \text{ consistent with } \mathcal{I}] \Rightarrow Q(U(D)) = V$$

□

First, note that since the view-independent-of-update question is a special case of the self-maintainability question, we do not absolutely need to address it in order to maintain the views in a data warehouse. We mention it here only for the sake of completeness and because having efficient methods to decide it may be valuable from a practical standpoint. Therefore, the view-independent-of-update question will not be explicitly addressed in this thesis, except that we will mention some of our own work that is closely related.

1.3.2 The Problem Dimensions in Runtime View Self-Maintenance

The problem of runtime view self-maintenance, as defined previously, has several parameters that can be classified into roughly three dimensions:

- Complexity of query Q that defines the view V to maintain, and of the queries that define any other views in the warehouse that are useful in the maintenance of V.
- Nature of the update U on the base relations sent by the data sources to the warehouse.
- Information available to the warehouse that can be used to perform its own maintenance. As mentioned before, we assume this information includes at the very least the instance of the view to maintain, its definition, and the update instance. In the following, we will only describe the additional information.

Complexity of View Definitions

Complexity of queries that define the views has a direct impact on the solution complexity. In this thesis, we mainly focus on the class of conjunctive queries, and will also touch on unions of conjunctive queries. Below, we give a brief description of these query classes. For a more detailed definition of these query classes and other relevant classes, we refer the reader to [Ull89]. In Section 2.1, we will also provide a more detailed description of these query classes.

Given a set of relations, a *conjunctive query* essentially looks for a particular pattern of tuple combinations that are present among the relations. In relational algebra terminology, a conjunctive query is a Select-Project-Join query where the selection conditions are restricted to equality comparisons and the joins are restricted to equijoins. Consider for example a database with the following three relations used by our retail chain TMart:

$sales(C, I)$:	indicates customer C bought merchandise item I
$cust(C, A)$:	indicates customer C resides in area A
$carry(M, I, A)$:	indicates competitor M carries merchandise item I in area A.

Consider the conjunctive query that asks for customers who bought products that are also carried by *vmart* (a competitor of *tmart*) in the same area as the customer's residence. In relational algebra notation, we write the query as

$$\pi_C\ (sales \bowtie cust \bowtie \sigma_{M=vmart}\ carry)$$

Let us now use this example query to introduce the *Datalog rule notation*, which we will use throughout this thesis. In Datalog, we write the query as the following single *rule*:

$$v(C) :\!-\ sales(C, I)\ \&\ cust(C, A)\ \&\ carry(vmart, I, A)$$

where $v(C)$ is called the rule's *head*, and $sales(C, I)$, $cust(C, A)$, and $carry(vmart, I, A)$ the rule's *subgoals*. The *body* of the rule refers to all the subgoals. The predicates that appear in the subgoals, namely *sales*, *cust*, and *carry*, represent the base relations. The predicate that appears in the head, v, is called the *query predicate*. The rule is also said to define predicate v. Note that the query predicate, which is used to return the answer to the query, is distinct from the predicates used in the rule's body. By convention, predicate names and constant values (e.g.. *vmart*) are written in lower case, and variables are written with their initial in upper case (e.g., C, I, and A). A subgoal specifies a set of tuples in the relation for the subgoal's predicate we are looking for. For instance, the subgoal $carry(vmart, I, A)$ looks for all tuples in the *carry* relation with value *vmart* in their M component. A variable that appears in two different subgoals represents an equijoin. For instance, the two occurrences of C among subgoals $sales(C, I)$ and $cust(C, A)$ indicate the requirement that the C component of a *sales* tuple must agree with the C component of a *cust* tuple. Finally, any variable in the rule's head must appear in the rule's body. A rule with this property is said to be *safe*.

The following lists additional restrictions on conjunctive queries that are sometimes applied in the thesis. These restrictions are important since we can often exploit them to find efficient solutions to the runtime view self-maintenance problem.

- *No Self-joins:* a conjunctive query has self-joins when more than one subgoal in the query's body use the same predicate. For instance, in the following query that asks for pairs of customers living at the same address, predicate *cust* is used more than once:

$$v(C, C') :\!-\ cust(C, A)\ \&\ cust(C', A)$$

- *No Projections:* a conjunctive query has projections when some variable in the query's body does not occur in the query's head. For instance, variable A in the previous query is projected out from the head of the query.

A query that is a *union* of conjunctive queries can be represented in the Datalog notation by a collection of one or more rules that define the query predicate. For instance, The following query, which asks for customers who either bought a television set or bought merchandise that "vmart" also carries, is an example of a union of conjunctive queries:

$$\begin{array}{lll} v(C) & :\!- & sales(C, I)\ \&\ carry(vmart, I, A) \\ v(C) & :\!- & sales(C, television) \end{array}$$

Nature of Base Updates

In this thesis, we only consider changes to the contents of the base data, even though changes to the base data schema also occur in practice. Further, the base updates are assumed to be ground, that is, updates consist of ground facts that are to be deleted from or inserted to the base relations. Base updates can be one of the following:

- Single updates (deletions or insertions).
- Multiple updates to a single base relation (deletions, insertions, or both).
- Updates to multiple base relations.

While single updates are easier to handle than multiple updates, multiple updates are important because there are situations in which a view can be maintained under multiple updates but not under each individual update.

In the following, we describe the various kinds of information that is available besides the definition and the instance of the view to maintain. Generally, information helps self-maintainability: the more information we can use to maintain a view, the more situations the view can be maintained. But using additional information may make the solution derivation and the solution itself more complex.

Integrity Constraints on the Base Relations

Base relations may be expensive to use for maintaining the views in a warehouse, but integrity constraints on the base relations are virtually free and often available. The following lists integrity constraints that are commonly found in database systems:

- Key constraints: given a relation, a key constraint on the relation specifies a set of attributes over which any pair of different tuples must disagree. In other words, the values of the attributes in the set uniquely identifies tuples in the relation.
- Functional dependencies: given a relation, a functional dependency, written as $\alpha \rightarrow \beta$, specifies two sets of attributes, α and *beta*, such that any pair of tuples that agree over α must also agree over β. Clearly, a key constraint is the special case of a functional functional where β represents all the attributes for the relation.
- Inclusion dependencies: given two relations, an inclusion dependency specifies two sets of attributes, one for each relation, such that for any tuple in the first relation, there must be a tuple in the second relation such that the both tuples agree over their respective sets of attributes.

This thesis only considers functional dependencies, which subsume the key constraints. Surprisingly, despite the local nature of functional dependencies (i.e., they only apply to individual relations), they do help in the view self-maintenance problem.

Single vs. Multiple Views

Data warehouses rarely consist of only one view. Further, given a view to maintain, it is often the case that there is another "related" view in the warehouse. Two views are related if either they derive from a common base relation or there is a third view related to both views. Related views should in principle help in maintaining the original view, since it may tell us more about the base relations.

Thus, even though single views (i.e., views that are not related to any other views in the warehouse) are simpler to deal with, it is important to also consider the case of multiple views since they are available locally at the warehouse for use in maintenance.

No Base Relations vs. Some Base Relations

Without using all base relations, there is no guarantee that a view will always turn out to be self-maintainable under some update. In fact, a view may not be self-maintainable even if we use all the information that is cheaply available to us, including for instance integrity constraints and additional views.

While base access should be avoided as much as possible, we must also have provisions for allowing access of base relations for view maintenance. Thus, any subset of the base relations is an additional piece of information we can use to maintain the warehouse.

Partial Copies

One form of partial knowledge of the base data is to have a subset of the base relations available. Another form of partial knowledge is when the contents of a given relation are not totally known: certain tuples are known to be in the relation and other tuples not to be in. Such information can be easily obtained from a log of the most recent changes to the base relations we keep at the warehouse. Since the contents of the base relations may not be entirely known, the information we have about them is called *partial copies*. For instance, suppose the following sequence of most recent changes to relation R can be extracted from the log:

$$\text{insert } R(2), \text{insert } R(3), \text{delete } R(2).$$

We do not know the exact contents of R, but we can be sure that:

- $(2) \notin R$
- $(3) \in R$.

Figure 1.9 summarizes the dimensions that are important to consider in the problem of runtime view self-maintenance for a data warehouse.

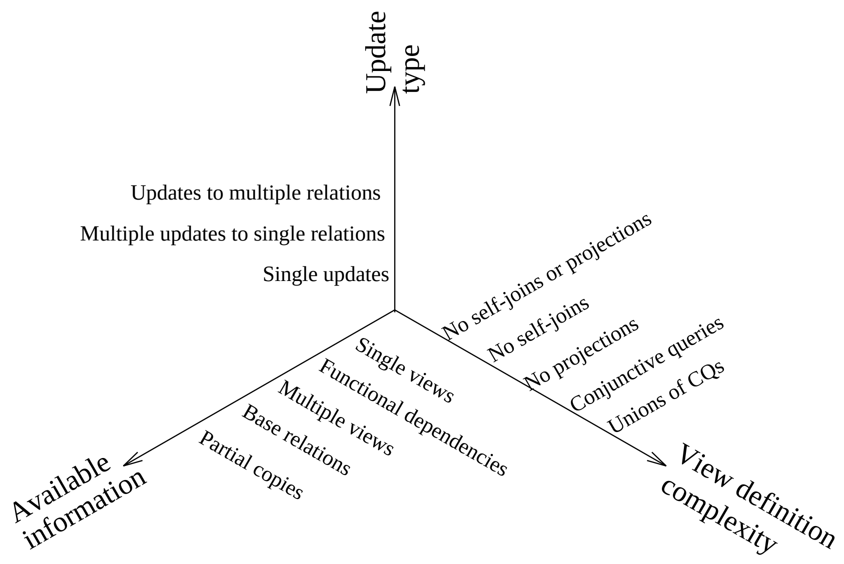


Figure 1.9: Multidimensional problem space in runtime view self-maintenance.

1.4 Contribution of this Thesis

1.4.1 Desiderata

Driven by both practical considerations and theoretical interests, we are looking for solutions to the runtime view self-maintenance problem that have the following desirable properties, which we define shortly:

- Solutions must be sound and complete.
- Solutions are generated at compile-time if possible.
- Solutions are in declarative form.
- Solutions must be as efficient as possible.

These constraints make the problem considerably more challenging. However, satisfying them all may not always be possible. The emphasis in this thesis is on retaining solution completeness, as defined below.

Solutions that are Sound and Complete

The main goal of view maintenance is to keep the views in the warehouse consistent with the base relations from the data sources. Thus, applying a solution to the problem must always result in correctly updating the views. In the following, we describe the specific requirement for answering each of the main questions:

- Self-Maintainability: a self-maintainability test that evaluates to *true* must guarantee that a view is indeed self-maintainable. Otherwise, we would be led to believe it is safe to apply a maintenance procedure to the view, and this application may result in incorrectly updating the view even though the maintenance procedure itself is correct when the view is self-maintainable.

- Maintenance: a maintenance procedure, when applicable (e.g., when the view is self-maintainable), must always result in correctly updating the view. That is, the resulting view should not contain more tuples than it ought to, or less.

- View independent of update: a test that is not sound would erroneously conclude that the view remains consistent with the new database state. And since no maintenance action follows, the view may become inconsistent with the new database state.

Solution completeness, by contrast, only applies to the self-maintainability and view-independent-of-update questions. When a complete test evaluates to *false*, we can be assured that the view is not self-maintainable and, without additional information, there is no way to maintain the view correctly. While the use of an incomplete test is not catastrophic, it may lead us to miss situations where a view is actually self-maintainable and to consume additional resources unnecessarily when trying to maintain the view (e.g. by accessing the base data).

Let us emphasize that the notion of completeness is relative to $\mathcal{I}$, the information we have available to maintain the views. The more information we have, the more situations we should be able to detect where a view is self-maintainable. If we choose to ignore some additional information, an incomplete test may result, even though the test itself is complete in the absence of the additional information.

To some extent, the notion of completeness is also relative to where we are along the dimensions other than the information dimension. For instance, a self-join can be viewed as joining different relations that have similar contents. Ignoring self-joins (i.e., treating different occurrences in a view definition of the same predicate as different predicates) amounts to ignoring this similarity constraint, and may lead to an incomplete solution. Also, to determine self-maintainability under a batch update by determining self-maintainability under individual updates may lead to an incomplete solution. In fact, as we show later in the thesis, there are situations where a view is self-maintainable under a batch update but not under some constituent update.

Solutions Generated at Compile Time

Runtime view self-maintenance does not mean that determining self-maintainability of a view or that determining how to maintain the view must be entirely performed at runtime. In fact, we can conceive of generating, at compile time, a view self-maintainability test and a view maintenance expression. This generation is based on which view we would like to maintain, its definition, an update type, and any additional information available at compile time such as constraints on the base relations, which other views will be available,

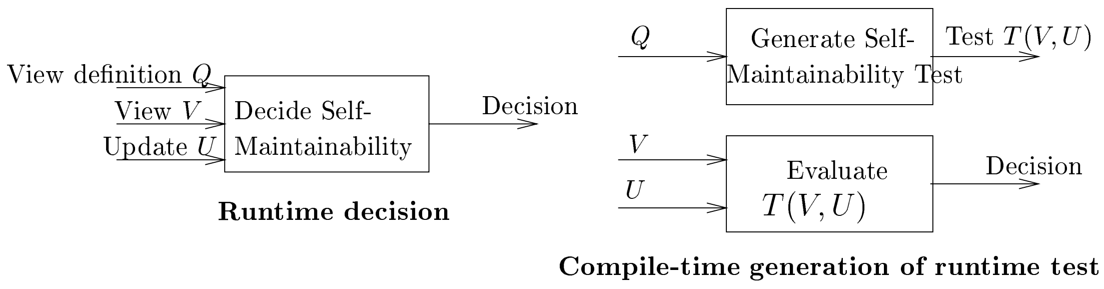


Figure 1.10: Separating test generation from test evaluation.

and which base relations we plan to use. Figure 1.10 illustrates this concept of compile-time generation of runtime solution for the problem of deciding self-maintainability.

While both the solutions in both approaches may have the same theoretical runtime complexity, the practical advantages of generating the solution at compile time are that:

- The same solution can be reused across multiple invocations of incremental view maintenance, instead of being rederived at every invocation.
- Expensive optimization can be applied to the solution at compile time, thereby minimizing the amount of work that needs to be performed at runtime to evaluate the solution.

Solutions in Declarative Form

Efficiency of runtime view self-maintenance can be further enhanced if the solutions can be expressed in declarative form, rather than in procedural form. Thus, the main practical benefits of having self-maintainability tests and maintenance expressions in the form of queries are that:

- We have more opportunities to simplify and optimize the solutions using known query optimization techniques.
- Not only we can take advantage of indexes defined on the views to speed up evaluation of the solution, but also we can use conventional query evaluation engines to execute the self-maintainability tests and the maintenance expressions.

Efficient Solutions

Finally, in order to justify using runtime view self-maintenance as a better alternative approach to maintaining data warehouses than using full base relations lookup, the solutions must be efficient: they must be efficient to generate and, even more importantly, efficient to evaluate at runtime.

1.4.2 Exclusions

This thesis works through only an important slice of the runtime view self-maintenance problem. Many other slices simply fall outside the scope of the thesis. In the following, we briefly describe some of the possibilities and issues we are not directly addressing in this thesis.

Warehouse Design for Self-Maintainability

Our work focuses directly on the problem of maintaining a *given* warehouse rather than the problem of *designing* a warehouse for better self-maintainability. To some extent, the results presented in this thesis can be applied to the design problem in a few cases:

- By analyzing the self-maintainability conditions we obtain for different view definitions, we can formulate simple design principles such as: when defining a view, avoid projecting out certain attributes without also projecting out certain other attributes (see Chapter 3).
- We can minimize views to be materialized at the warehouse by comparing self-maintainability conditions obtained for a given view collection and for a smaller collection. If the two sets of conditions are equivalent, then the smaller collection of views is as self-maintainable as the larger one.
- We may be able to compare two warehouse designs (i.e., views in the warehouse) by comparing their self-maintainability conditions. If the first set of conditions subsumes the second set, then the first design is definitely more self-maintainable than the second design.

Even though our results can be used to help make design decisions in some cases, more powerful analytical tools are needed. For instance, if two self-maintainability conditions are not comparable (i.e., no one subsumes the other), we currently do not have a way to quantify the degree of self-maintainability of a view collection.

Which Base Relation Subset to Use

When we allow the warehouse maintenance system to draw on the external sources to maintain the views at the warehouse, the question as to which subsets of the base relations to consider can be raised. This question is not addressed in the thesis. A good heuristics may consist of considering those base relations with the lowest access cost first. However, finding an optimal plan for choosing different subsets of base relations to consider next is a more difficult problem. To implement it properly may also require some measure of confidence that a given subset of base relations would give us enough information to succeed at the next round.

Trading Off Completeness for Efficiency

Requiring completeness in the solutions assures us that we will not miss any opportunity to maintain a view successfully. As we mentioned earlier, the consequence of missing some self-maintainable situations may force us to fetch more information from the external data sources, and to incur higher costs. But without being able to quantify the costs associated with these external accesses, we cannot understand how to trade off completeness of self-maintainable tests for their execution speed. Further, we also need to more precisely measure how close from completeness a given test is.

Intensional Updates

In this thesis, we assume the updates that the data sources send to the warehouse are ground. That is, these updates consist of ground tuples to be deleted from or inserted into the base relations. While this situation is very common in practice, there are cases where the tuples to be deleted or inserted can be more naturally and more concisely specified as a query. We call such updates *intensional updates.*

Concurrency Control

If we allow the warehouse maintenance system to access some of the base relations either in evaluating self-maintainability of a view or in applying a maintenance expression to a view, we must be careful if we also allow other modifications to the base relations to be made concurrently. For instance, we already mentioned the view update anomaly problem associated with the approach of view maintenance with unrestricted source access, and work ([Z*95]) that deals with this problem. However, it is still not clear how to extend the techniques from [Z*95] to adjust the self-maintainability decision.

1.4.3 Related Work

The traditional problem of incremental view maintenance, that is, with unrestricted access to the base relations, has been well studied. We briefly describe only a few of the key papers. [BC79, QW91, CW91, GLT97] studied the incremental view maintenance problem for views defined by nonrecursive queries. The problem for first-order queries and recursive queries was considered in [DS92, DT92]. [Kuc91, HD92, UO92, GMS93] studied the maintenance of recursive views, and [SJ96] considered the same problem but without using the materialized views. Concurrency issues in view maintenance are treated in [Z*95, ZWG97]. Techniques for incremental view maintenance based on counting were developed in [SI84, BLT86, GKM92, GMS93], and techniques based on algebraic differentiation of view expressions can be found in [Pai84, QW91, SJ96, GLT97].

By contrast, limiting base data access opens up new dimensions to the maintenance problem which still remain largely unexplored. [GM95] gave an excellent taxonomy for the different types of information available for view maintenance.

Since the notion of runtime view self-maintainability is at the heart of this thesis work, we will first describe related work based on this notion. Although the notion of compile-time view self-maintainability is not directly relevant to our work, it can provide us with a better understanding of the runtime notion. For this reason, we will also mention work based on the compile-time notion. Finally, we describe work on checking global integrity constraints in distributed databases and discuss how it relates to the problem of view self-maintenance.

Work Based on Runtime Self-Maintainability

The problem of incrementally maintaining a view without using any base relations was first studied in [TB88, BCL89], which addressed the question of detecting conditionally autonomously computable updates. The effect of an update on a view is said to be *conditionally autonomously computable* if there exists a function that depends only on the view instance, the update instance, and the view definition, and that computes the new state of the view from its current state regardless of the underlying database. Clearly, a view is runtime self-maintainable under an update if the effect of the update on the view is conditionally autonomously computable. The converse, however, is not known to hold in general for arbitrary view definitions. While it is not clear to what extent the notion of runtime of view self-maintainability is more general than the notion of conditional autonomous computability, we would like to point out that at least within the scope of this thesis (where the view definitions are restricted to unions of conjunctive queries with arithmetic comparisons), the two notions can be shown to be equivalent.

The problem of runtime view self-maintenance considered in [TB88] has the following restrictions:

- Views are defined by conjunctive queries with arithmetic comparisons but no self-joins.
- Updates are either tuple insertions or deletions to a single relation. Arbitrary updates to multiple relations are not considered.
- No additional information besides the view itself and the update is used in maintaining the view. In particular, their work did not consider exploiting multiple views, dependencies on base relations, or a subset of the base relations.

The solution for the self-maintainability question, given in [TB88], consists of building, at runtime, a formula for the self-maintainability condition. As given, the solution has the following drawbacks:

- The issue of how to check the validity of the formula and the complexity of the problem were not addressed.
- The formula is an expression that uses the contents of the view instance. The size of the formula is thus a function of the size of the view instance, which makes its practical implementation difficult.
- The solution did not show how to maintain the view when the view is self-maintainable.

[GB95] later found the solution given in [TB88] to be incorrect and subsequently corrected it. However, the solutions given in [GB95, TB88] remain difficult to implement efficiently, and their method cannot be extended easily to take advantage of multiple views, dependencies on base relations, and partial base access. Our own work on runtime view self-maintenance can be found in [Huy96a, Huy96b, Huy96c, Huy97b]: [Huy96a] considered the problem for single CQ views with no self-joins; [Huy96b] extended the results to the class of CQ views with a restricted form of self-joins; [Huy96c] studied the problem in the presence of functional dependencies; and [Huy97b] treated the general problem for CQ views in the presence of FD's and under arbitrary ground updates.

Work Based on Compile-Time Self-Maintainability

The notion of compile-time self-maintainability also has its root in [BLT86, TB88, BCL89] where they called it *Unconditional Autonomous Computability.* [BCL89] essentially shows that most views, except for the very simple ones that do not use joins, are not self-maintainable under insertions. The significance of this result is that without considering using additional information such as integrity constraints on the base relations, the compile-time self-maintainability notion has very limited applicability. Subsequently, [GJM96] showed that compile-time self-maintainability can be improved when key constraints on the base relations are considered. [JMS95] studied the self-maintenance problem where all the base relations are available except the ones that are being updated. More recently, [Q*96, Qua97] tackled the warehouse maintenance problem from a new angle, that of making a view self-maintainable by materializing a set of auxiliary views that facilitate self-maintenance. [Q*96] took advantage of key constraints and referential integrity constraints on the base relations to keep the auxiliary views small. However, they did not address the problem of making the warehouse self-maintainable starting from a warehouse with multiple materialized views.

View Independent of Update

As mentioned earlier, one of the core questions in runtime view self-maintenance is the question of view independent of update: given a view instance and a base update instance, does the view remain consistent with the updated base relations? This question turns out to be a special case of a known problem, the problem of checking global integrity constraints in distributed databases.

In this problem, we are given a collection of relations, an integrity constraint on the relations, and an update on the relations. If the relations are known to satisfy the constraint before the update, how can we guarantee that they continue to satisfy the constraint after the update? The problem becomes nontrivial when not all the relations are available for use.

EXAMPLE 1.4.1 Consider a university information system with the following three relations distributed across different sites:

$enroll(Student, Course)$
$prerequisite(Course, Course)$
$took(Student, Course).$

The enrollment policy that a student may enroll in a course only if he has taken all the prerequisites for that course, can be expressed by the following integrity constraint:

$$:\!-\ enroll(S,C)\ \&\ prerequisite(C,C')\ \&\ \neg took(S,C')$$

Suppose we would like to enroll *Smith* in *CS420*. Without looking at the relation for *prerequisite*, how can we be sure that *Smith*'s enrollment in *CS420* will not violate the enrollment policy? The answer is to make sure there is a least one student who has enrolled in *CS420* successfully, and *Smith* has taken any course that each and every such student has taken. This solution is the most general possible and can be derived from the method developed in [Huy97a, Huy97c]. □

Given a view V, a query Q over database D that defines the view, and a base update U, the question of view independent of update in runtime view self-maintenance can be casted as a question of checking global integrity constraints in distributed databases if we treat the relationship $Q(D) = V$ as an integrity constraint between the view relation V and the base relations in D. Thus, the fact that view V remains consistent with the updated database is just another way of saying that V and $U(D)$ satisfy the integrity constraint $Q(U(D)) = V$.

Integrity constraints generally can be expressed as queries for violations. In this notation, the enrollment policy in the example above is written as the query:

$$panic :\!-\ enroll(S,C)\ \&\ prerequisite(C,C')\ \&\ \neg took(S,C')$$

where *panic* is a special 0-ary predicate. This query asks for the existence of integrity-constraint violations. In the following discussion, the class of a constraint refers to the class of the query representing the constraint in this notation.

[G*94] solves the integrity constraint checking problem

- For the class of constraints that are unions of conjunctive queries with arithmetic comparisons,
- For insertions into single relations, and
- When the updated relation is available for use in the check.

Now consider the question of view independent of update in which:

- Q is a union of conjunctive queries with arithmetic comparisons,
- U is a set of insertions into some base relation,
- The updated base relation is available for use in answering the question.

This question can be rephrased as that of checking constraint $Q(D) = V$. But since Q is a monotonic function of the base relations, this constraint can be reduced to $Q(D) \subseteq V$, which is essentially a union of conjunctive queries with arithmetic comparisons (since negation applies only to the predicate for the view relation whose instance is available). The results from [G*94] can thus be adapted to answer the view independent of update question.

However, when we consider deletions to the base relations, the constraint we must enforce becomes $Q(D) \supseteq V$, which contains negation that applies to relations whose instances may not be available. To answer the question of view independent of deletion using the constraint checking method requires treatment of constraints involving negation. Such treatment can be found in [Huy97a, Huy97c], in which we solved the integrity constraint checking problem

- For the class of constraints that are conjunctive queries with negation,
- Under arbitrary sets of insertions or sets of deletions,
- And where the updated relation is available for use in the check.

The following is an example of the view independent of deletions problem that can be resolved using the results from [Huy97a].

EXAMPLE 1.4.2 Consider a view V defined by

$$\begin{array}{lll} v(X) & :- & a(X) \ \& \ b(X) \\ v(X) & :- & b(X) \ \& \ c(X) \end{array}$$

where A, the relation for a, is the only base relation available for use in testing whether V remains consistent after a tuple is deleted from A. Since no changes are made to the relation for b, the problem of determining whether the view is not affected by a deletion to A reduces to checking if a deletion to A preserves consistency under the constraint:

$$panic :- v(X) \ \& \ \neg a(X) \ \& \ \neg c(X)$$

whose solution consists of checking if the tuple to be deleted is not present in both V and A. This test is both sound and complete, and can be obtained from the general results from [Huy97a]. □

Before closing this section on related work, we would like to mention work on *query-independent-of-update*, a problem closely related to the question of view-independent-of-update. In this problem, we are given a query and a base update. We would like to decide whether or not the query gives the same answer both before and after the update, for all instances of the base relations. Since the contents of the answer to the query prior to the update are not used in the decision, this problem can be viewed as the compile-time analog of the view independent of updates problem.

The query-independent-of-update problem was originally considered in [BC79, BLT86, TB88, BCL89]. The problem was called *detecting irrelevant updates* in [BCL89], and was solved there for the class of conjunctive queries with no self-joins. [Elk90] later extended the results to queries that are nonrecursive Datalog programs where the updated predicate is not repeated. More recently, [LS93] solves the query-independent-of-update problem for queries that are recursive Datalog programs with arithmetic comparisons and stratified negation.

1.4.4 Results

Figure 1.11 summarizes the results achieved in this thesis work. In Figure 1.11, the numbers in the table cells correspond to the number of the chapters where the results will be presented. The three levels of shading denote the different levels of runtime complexity of our solutions: the dark shaded solutions are exponential in the size of the view instance, the medium shaded ones are polynomial, and the light shaded ones are linear.

1.5 Outline of The Thesis

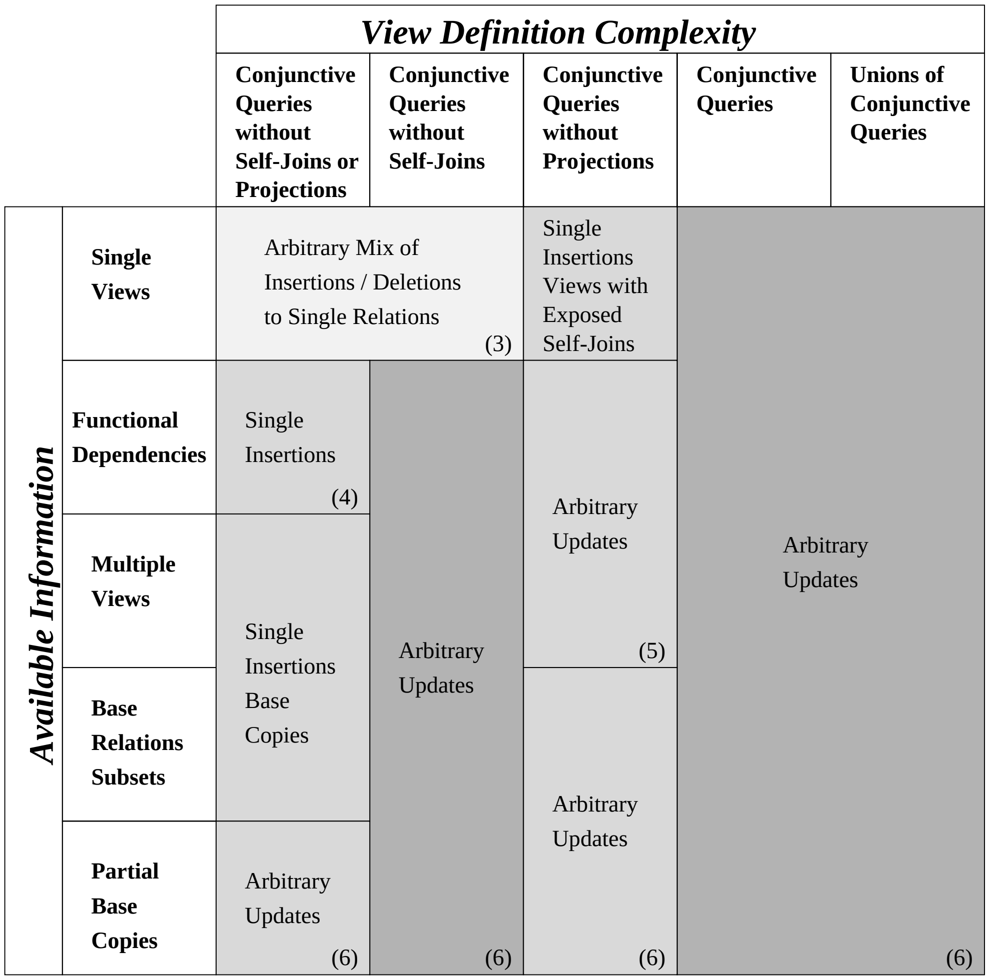
Chapter 2, *Preliminaries,* begins with a description of the notational conventions used throughout this thesis. We then present some of the basic concepts and properties that are useful in developing solutions for the runtime view self-maintenance problem in the subsequent chapters: canonical databases and database consistency. Finally, we introduce the notion of instance specific query containment and the problem of translating a query containment decision to a query, which underlie the general technique used in Chapter 5 to solve the generalized view self-maintenance problem.

Chapter 3, *Strict View Self-Maintenance,* gives a full treatment of the runtime view self-maintenance problem for single views defined by conjunctive queries with no self-joins. We call self-maintenance in this case strict because we use no information besides the view to maintain and the base update. We show that strict view self-maintenance admits very efficient solutions.

Chapter 4, *Exploiting Functional Dependencies,* considers using functional dependencies in helping self-maintain views in the absence of additional information. We show that these dependencies generally can be used to improve view self-maintainability. Even under general functional dependencies, we can obtain a simple characterization of the solutions for maintaining, under single insertions, single views defined by conjunctive queries with no self-joins.

Chapter 5, *Generalized VSM for Views with no Projections,* generalizes strict VSM (view self-maintenance) to include using a combination of the following features: multiple views, functional dependencies, and arbitrary updates. To solve the generalized view self-maintenance problem, we develop a method based on the notion of instance specific query containment explained earlier in Chapter 2. This chapter addresses the problem for views defined as conjunctive queries with no projection.

Chapter 6, *Extensions,* discusses extending the results from Chapter 5 to cover the use of projections and unions in the view definitions, and the additional use of subsets and



		View Definition Complexity				
		Conjunctive Queries without Self-Joins or Projections	**Conjunctive Queries without Self-Joins**	**Conjunctive Queries without Projections**	**Conjunctive Queries**	**Unions of Conjunctive Queries**
Available Information	**Single Views**	Arbitrary Mix of Insertions / Deletions to Single Relations (3)		Single Insertions Views with Exposed Self-Joins	Arbitrary Updates (6)	
	Functional Dependencies	Single Insertions (4)	Arbitrary Updates (6)	Arbitrary Updates (5)		
	Multiple Views	Single Insertions Base Copies				
	Base Relations Subsets			Arbitrary Updates (6)		
	Partial Base Copies	Arbitrary Updates (6)				

Figure 1.11: Runtime view self-maintenance results at a glance.

partial copies of the base relations in solving the generalized view self-maintenance problem.

Chapter 7, *Conclusion,* summarizes the main contribution of the thesis and discusses possible avenues for future work on runtime view self-maintenance.

Chapter 2

Preliminaries

In this chapter, we describe the notation, state the general assumptions, and define the basic concepts and techniques that will be used throughout this thesis. This chapter is organized as follows:

Section 2.1, ***Notation and Assumptions,*** describes the Datalog notation we use in this thesis to represent the queries involved in the view definitions, self-maintainability conditions, maintenance expressions, and also in the process developing the solutions.

Section 2.2, ***Canonical Databases,*** defines a concept that is fundamental in the problem of view self-maintenance.

Section 2.3, ***Database Consistency,*** defines consistency of database instances relative to a given view instance and describes consistency of canonical databases based on the assumption that the given view is realizable.

Section 2.4, ***Rectifying Conjunctive-Query Views With No Self-Joins,*** describes a very important property for the special case of views defined as conjunctive queries with no self-joins that allows us to find simple solutions to the view self-maintenance problem.

Section 2.5, ***Query Containment,*** describes the query containment problem, which will serve as the basis for our method for solving the view self-maintenance problem in the more general cases. In particular, we describe the concept of expressing instance-specific query containment as an efficient query.

2.1 Notation and Assumptions

In this thesis, we assume a relational database framework in which views are defined by relational queries over base relations. Set semantics is also assumed. Thus, the answer to a query is a set of tuples.

We will use the notation of Datalog [Ull89] both for defining views and for representing the queries involved in our algorithms. This choice is by convenience, even though any other relational languages could be used. Thus, the view definition Q for view V from Example 1.2.3 is written in the Datalog rule notation as the following single rule:

$$v(X, Y, Z) \text{ :- } r(X, Y) \ \& \ s(X, Z) \ \& \ t(Y, Z)$$

where $v(X, Y, Z)$ is called the rule's *head* and $r(X, Y)$, $s(X, Z)$, and $t(Y, Z)$ are the rule's *subgoals*. By convention, relation names are written in upper case (e.g., V, S, and δR) and their predicate in lower case (e.g., v, s, and δr). The *extension* of a predicate is the instance of the relation for the predicate. In general, a predicate is called an *IDB predicate* if it appears in the head of some rule, an *EDB predicate* otherwise. A particular IDB predicate that is used to return the answers to the query is called the *query predicate*. Thus, in query Q above, predicates r, s, and t are the EDB predicates, and v the query predicate. A query defined by a single rule whose body contains only EDB subgoals, such as Q above, is called a *conjunctive query* (see [CM77]). In relational algebra, conjunctive queries correspond to Select-Project-Join (or SPJ in short) queries with only equality comparisons.

In rule notation, we generally write a conjunctive query as

$$H \text{ :- } G_1 \ \& \ \ldots \ \& \ G_n$$

where constant symbols and variables are not explicitly spelled out. Constant symbols may appear anywhere in the rule, and we generally assume that each variable in the rule's head also appears in the rule's body. Such rule is said to be *safe* (defined based on [Zan86]).

In general, variables used in a conjunctive query's body do not need to appear in the query's head. A variable that does not appear in the head is said to be *hidden*. A variable that appears in the head is said to be *exposed*.

Conjunctive queries with no projections turn out to be an important class of queries in our thesis work. A conjunctive query is said to have *no projections* when all the variables in the query are exposed.

Another important subclass of conjunctive queries consists of conjunctive queries with no self-joins. A *self-join* is a pair of subgoals in the rule's body with the same predicate.

There will be occasions where queries more complex than conjunctive queries are used, albeit without recursion.

A union of conjunctive queries (see [SY80]) is like a conjunctive query except that we have several rules defining the query predicate. We will use H :- $A \mid B$ as a shorthand for the two rules H :- A and H :- B.

A conjunctive query with negation (see [LS93] for example) is defined by a rule that has at least one negated subgoal. We assume that each variable used in a negated subgoal also appears in some non-negated subgoal. Such a rule is said to be *safe* (see [Zan86]).

A conjunctive query with arithmetic comparisons (see [Klu88]) is defined by a rule that has, besides *ordinary* subgoals (i.e., subgoals with a predicate), other subgoals of the form μ `op` ν where μ and nu are variables or constant symbols, and `op` is one of the operators $<$, $\leq$, $>$, $\geq$, and $\neq$. These subgoals are called arithmetic comparisons. We generally assume that a rule with arithmetic comparisons is safe, that is, each variable used in an arithmetic comparison also appears in some ordinary (non-negated) subgoal.

A nonrecursive Datalog query with negation (see [LS93] for example) is defined by a collection of rules that may have negated subgoals in their body and such that no predicate depends on itself. A predicate p is said to *depend on* predicate r (which may or may not be identical to p) if either p is defined by a rule whose body uses predicate r or p is defined by a rule whose body uses some predicate q that depends on r. This class of queries corresponds to the entire class of relational algebra queries.

For more details on the Datalog notation and other classes of queries, see [Ull89, AHV95].

There will be numerous occasions where the variables used in a rule need to be made explicit. In the following rule for instance,

$$v(\bar{X}', \bar{Y}', \bar{Z}') \text{ :- } r(\bar{X}, \bar{Y}) \ \& \ S(\bar{Y}, \bar{Z}),$$

we use $\bar{X}'$, $\bar{Y}'$, $\bar{Z}'$, $\bar{X}$, $\bar{Y}$, and $\bar{Z}$ to denote *ordered sets* or vectors of variables. We often refer them as sets, with the implicit assumption that the elements are ordered. In the rule body, $S(\bar{Y}, \bar{Z})$ denotes either a single subgoal or a conjunction of subgoals that uses variables $\bar{Y}$ and $\bar{Z}$. In any case, we will always specify what S represents to avoid confusion. We will also make the relationships between different sets of variables explicit, as the need arises. For instance, if the rule above represents a conjunctive query, we may specify that $\bar{X}$, $\bar{Y}$, and $\bar{Z}$ represent disjoint sets of variables. Furthermore, to express the fact that the rule is safe, we will say that $\bar{X}'$, $\bar{Y}'$, and $\bar{Z}'$ are subsets of $\bar{X}$, $\bar{Y}$, and $\bar{Z}$ respectively. Unless indicated otherwise, for any capital letter X, we will use $\bar{X}'$ to denote a subset of $\bar{X}$, as a convention. Note that the use of this set-of-variable notation does not imply a particular order of occurrence of the variables in the subgoals, but only some fixed order. Nor does it imply that each variable in a set only occur once in a subgoal, a conjunction of subgoals, or in the head. Finally, the absence of constant symbols in this notation does not preclude the use of constant symbols in the rule.

Suppose r is the predicate for relation R to which we would like to insert some tuple. We say for example that we want to insert $r(\bar{a}, \bar{b})$. In this notation, for any lower-case letters a and b, $\bar{a}$ and $\bar{b}$ denote vectors of constants that match the vectors $\bar{X}$ and $\bar{Y}$ respectively. Also, for any lower-case letters u and v and for any capital letter X, given two vectors of constants $\bar{u}$ and $\bar{v}$ and a set $\bar{X}$ of variables, we will use $\bar{u} =_{\bar{X}} \bar{v}$ to denote the fact the two constant vectors agree over the variables specified by $\bar{X}$. $\bar{u} \neq_{\bar{X}} \bar{v}$ denotes the fact that they disagree.

2.2 Canonical Databases

In this section, we consider a simple view self-maintenance problem where:

- We are given only one view to maintain and to use.
- The view is defined by a conjunctive query over the base relations.

and define the concept of canonical databases in this simple context. In later chapters, this concept will be generalized for more complex contexts. However the basic idea remains the same.

Definition 2.2.1 (**Canonical Database**) Let V be a view, and let Q be the conjunctive query that defines V. The *canonical database*, denoted $\hat{D}$, consists of all the tuples obtained as follows: for each tuple in V, the matching of the tuple with Q's head provides a substitution for the variables in Q's body that appear in the head; this substitution is extended to the remaining variables by binding each of them to a *new symbol*; the ground atoms obtained after making this extended substitution into Q's body are included in $\hat{D}$. □

In this definition, it is important to note the following:

- Canonical databases are defined relative to both the view definition and the view instance.
- We have assumed that every tuple in the view matches the head of the rule defining the view. This property follows from the *view-realizability assumption* we will describe in the next section.
- Strictly speaking, canonical databases are not unique. But since they are all isomorphic to each other (i.e., identical up to renaming of the new symbols), we will use "the" canonical database to loosely denote an arbitrary one among all the isomorphic canonical databases.
- In the context of a single view V, we will also use $Q^{-1}(V)$ to denote the canonical database $\hat{D}$.

EXAMPLE 2.2.1 Consider the view definition

$$v(X,Z) \text{ :- } s(X,Y) \ \& \ s(Y,Z)$$

Consider the instance $V = \{(a_1, c_1), (a_2, c_2)\}$, shown in Figure 2.1. Since each tuple in V provides a substitution only for variables X and Z, the substitution must be extended to all variables by binding Y to a new symbol, i.e., a symbol that does not occur in the view definition or the view instance. Note that new symbols are created for Y for each tuple in V. In Figure 2.1, Y gets y_1 for the first V-tuple and y_2 for the second V-tuple. The canonical database $\hat{D}$ consists of $S = \{(a_1, y_1), (y_1, c_1), (a_2, y_2), (y_2, c_2)\}$. □

$v(X, Z)$	$s(X, Y)$	$s(Y, Z)$
a_1, c_1	a_1, y_1	y_1, c_1
a_2, c_2	a_2, y_2	y_2, c_2

Figure 2.1: A view instance and the associated canonical database.

Conceptually, we are trying to reconstruct the underlying database from a given view instance. But since it is generally not possible nor desirable to reconstruct the full database, we use the canonical database to capture sufficient information about the underlying database to allow us to analyze self-maintainability of the view.

As we will see in a moment, there are many cases where the canonical database is consistent with the view instance (at the end of this section, we will give an example showing that it is possible for a canonical database not to be consistent with the view). In such cases, we mainly use the canonical database as a reference database to compare the effect of updates on the view with other databases that are consistent with the view.

In practice, we do not intend to actually compute the canonical database. Instead, we use the definition of a canonical database as a tool to analyze the self-maintenance problem. In the simplest case, canonical databases are only used in proving the correctness of the solution. They do not appear in the solution itself.

2.3 Database Consistency

A view is defined by a query over a collection of base relations. These base relations are collectively referred to as the underlying database. Given an instance of the view, we are only interested in those database instances whose answers to the query are exactly the tuples in the view instance. These databases are said to be consistent with the view instance, as stated in the following definition.

Definition 2.3.1 (**Database consistent with a view**) Given a query Q that defines a view V in terms of a database D. A database (instance) D is said to be *consistent* with view (instance) V if $Q(D) = V$. □

Throughout this thesis, we assume that a given view instance V is the result of querying some database. In other words, we rule out any view instance that cannot be the result of querying any database. This situation may arise in practice if the view has been erroneously updated. This assumption is stated as follows:

Definition 2.3.2 (**View Realizability Assumption**) Given a query Q that defines a view V in terms of a database D. A view instance V is said to be *realizable* if there is a database instance D consistent with V, i.e., $V = Q(D)$. □

Under this assumption, we first state a lemma that relates a view with the canonical database, and then show two cases where the canonical database is guaranteed to be consistent with the view.

Lemma 2.3.1 *Let V be a view defined by a conjunctive query Q over some database. Let $Q^{-1}(V)$ be the canonical database. Then $Q(Q^{-1}(V))$ always contains V.* □

Proof: By construction of the canonical database, for every tuple t in V, there is a substitution that turns Q's head into t and Q's body into tuples that are in the canonical database $\hat{D}$. In other words, $t \in Q(\hat{D})$. ∎

We now show a situation where the canonical database is guaranteed to be consistent with the view.

Theorem 2.3.1 *Let V be a view defined by a conjunctive query Q over some database. Let $Q^{-1}(V)$ be the canonical database. If Q has no projection, then $Q^{-1}(V)$ is consistent with V.* □

Proof: Let D be the actual underlying database. When Q has no projection, $Q^{-1}(V)$ uses only constants from V and the tuples in $Q^{-1}(V)$ represent the ground facts that must be true of D in order to explain the tuples in V. In other words, $Q^{-1}(V) \subseteq D$. Since Q is monotonic, it follows that $Q(Q^{-1}(V))$ is contained in $Q(D)$, which is V. Since we also have $V \subseteq Q(Q^{-1}(V))$ (following Lemma 2.3.1), we conclude that $Q(Q^{-1}(V)) = V$. ∎

We now show another situation where the canonical database is also guaranteed to be consistent with the view.

Theorem 2.3.2 *Let V be a view defined by a conjunctive query Q over some database. Let $Q^{-1}(V)$ be the canonical database. If Q has no self-joins, then $Q^{-1}(V)$ is consistent with V.* □

Proof: Let D be the actual underlying database. First, to show that $Q(Q^{-1}(V)) \subseteq V$, we will show that any tuple in $Q(Q^{-1}(V))$ is also a tuple in $Q(D)$. First, let us write Q as H :- $G_1, \dots G_n$, and let us use the layout of the canonical database shown in Figure 2.2 for the rest of the proof. Figure 2.2 shows a view instance that consists of the tuples $t_1, \dots, t_m$ and the associated canonical database that contains the A_{ij} atoms. The atoms in each row corresponds to the view tuple in the same row. When the view definition uses no self-joins, different columns do not share atoms since atoms from different columns use different predicates.

Now, let $t \in Q(Q^{-1}(V))$. There is a variable substitution σ such that $t = \sigma(H)$ and $\sigma(G_j) \in Q^{-1}(V)$ for every j. In the absence of self-joins in the view definition, if follows that σ maps G_j to some atom in the same column as G_j. That is, $\sigma(G_j) = A_{i_j j}$ for some i_j, and furthermore:

$$\sigma \ : \ \begin{array}{l}\text{Hidden variables} \longrightarrow \text{New symbols} \\ \text{Exposed variables} \longrightarrow \text{View constants}\end{array}$$

H	G_1	$\ldots$	G_j	$\ldots$	G_n
t_1	A_{11}	$\ldots$	A_{1j}	$\ldots$	A_{1n}
$\vdots$	$\vdots$		$\vdots$		$\vdots$
t_i	A_{i1}	$\ldots$	A_{ij}	$\ldots$	A_{in}
$\vdots$	$\vdots$		$\vdots$		$\vdots$
t_m	A_{m1}	$\ldots$	A_{mj}	$\ldots$	A_{mn}

Figure 2.2: Canonical database for a view with no self-join.

Further, for each tuple $t_i \in V$, there must be some atoms in D that explain it. In other words, there is a symbol mapping h_i

- That maps the new symbols created in row i of the canonical database to some constants from D,
- That maps other non-new symbols to themselves,
- And such that $h_i(A_{ij}) \in D$.

Finally, consider a variable substitution σ' we construct from σ and the h_i's as follows:

$$\begin{array}{lll} \sigma' & : & \text{Hidden variables} \xrightarrow{\sigma} \text{New symbols} \overset{Some\ h_i}{\longrightarrow} \text{Constants in } D \\ & & \text{Exposed variables} \xrightarrow{\sigma} \text{View constants} \end{array}$$

It is easy to verify that for every j, $\sigma'(G_j) = h_{i_j}(\sigma(G_j)$. But $h_{i_j}(\sigma(G_j) = h_{i_j}(A_{i_j j}) \in D$. It follows that $\sigma'(G_j) \in D$ and $\sigma'(H) = \sigma(H) = t$ since H has no hidden variables. Therefore, $t \in Q(D) = V$, and $Q(Q^{-1}(V)) \subseteq V$. But since $Q(Q^{-1}(V)) \supseteq V$ (following Lemma 2.3.1), we conclude that $Q(Q^{-1}(V)) = V$. ∎

EXAMPLE 2.3.1 Consider the view definition

$$v(X,Y) \coloneq r(X,Y) \ \&\ s(Y,Z)$$

and the instance $V = \{(a_1, b), (a_2, b)\}$. The view definition has no self-joins. The canonical database is shown in Figure 2.3. Even though $r(a_1, b)$ joins with $s(b, z')$, and $r(a_2, b)$ joins with $s(b, z)$, only $v(a_1, b)$ and $v(a_2, b)$ can be derived. We cannot derive tuples other than those already in V. In other words, the canonical database is consistent with the view instance. □

However, when the conjunctive query that defines the view has both self-joins and projections, the converse of Lemma 2.3.1 is not true in general. As a result, the canonical database is not necessarily consistent with the view instance in general. The following example shows a view instance and a conjunctive query for which the canonical database is not consistent with the view.

$v(X,Y)$	$r(X,Y)$	$s(Y,Z)$
a_1, b	$r(a_1, b)$	$s(b, z)$
a_2, b	$r(a_2, b)$	$s(b, z')$

Figure 2.3: Canonical database consistent with the view from Example 2.3.1.

EXAMPLE 2.3.2 Consider the view definition

$$v(X) \coloneq s(X,Y) \ \& \ s(Y,X)$$

Note that this query uses the same predicate s twice in the query's body. Moreover variable Y does not appear in the query's head. Consider the instance $V = \{(a)\}$. The canonical database consists of $S = \{(a,y),(y,a)\}$ where y is a symbol other than a. Applying the query to this database results in $\{(a),(y)\}$. The canonical database is not consistent with the view instance. □

2.4 Rectifying Conjunctive-Query Views With No Self-Joins

When a view is defined by a conjunctive query with no self-joins, we claim that no generality is lost if we assume:

- The query uses no constants either in the head or the body.
- Within each subgoal in the query's body, variables occur only once.

In other words, constants and variable repetition within subgoals in the view definition play no role in the problem of view self-maintenance. We can then justify using a representation of the view definition, called the *rectified* representation, where only the pattern of variable sharing between the subgoals and the head matters. Thus, throughout the remaining of this thesis, a view definition with no self-joins will be represented as follows:

$$v(\bar{X}) \coloneq r_1(\bar{X}_1) \ \& \ r_2(\bar{X}_2) \ \& \ \ldots \ \& \ r_n(\bar{X}_n)$$

where v is the predicate for the view and for each $i = 1, 2, \ldots, n$, r_i is the predicate for base relation R_i. In this representation, $\bar{X}, \bar{X}_1, \bar{X}_2, \ldots, \bar{X}_n$ denote sets of variables, and $\bar{X}$ is a subset of the union of the $\bar{X}_i$'s.

Thus, the purpose of this section is to validate the claim that constants and variable repetition play no role in the view self-maintenance problem. Another purpose of this section is to show how straightforward it is to adapt the solutions to the self-maintenance problem given later in this thesis to arbitrary view definitions with no self-joins.

Let us first define, for a view definition with no self-joins, its rectified representation.

Definition 2.4.1 (**Rectified Representation**) Let view V be defined by conjunctive query $Q : H \coloneq G_1 \ \& \ \ldots \ \& \ G_n$, where H is the head with predicate v that uses variables $\bar{X}$, and for every $i = 1, \ldots, n$, G_i is a subgoal with predicate r_i that uses variables $\bar{X}_i$. Constant symbols may be used in Q, and within each literal in Q, variables may occur more than once. The *rectified representation* of $(v, Q, r_1, \ldots, r_n)$ is $(v', Q', r'_1, \ldots, r'_n)$, where the view predicate v' and the query Q' are defined as follows:

- View instance V' is defined in terms of V by: $v'(\bar{X}) = H$.
- View predicate v' is defined by query $Q' : v'(\bar{X}) \coloneq r'_1(\bar{X}_1) \ \& \ \ldots \ \& \ r'_n(\bar{X}_n)$.

□

In the rectified representation defined above, it is important to note the following:

- The insertion (resp. deletion) of a tuple t into (resp. from) R_i is represented by the insertion (resp. deletion) of a tuple t' into (resp. from) R'_i if G_i matches $r_i(t)$. In this case, t' consists of the constants in the bindings produced by the successful match.
- In query Q', although a variable occurs at most once *within* each subgoal, it may occur in more than one subgoal.

The following example illustrates the relationship between the original representation and the rectified representation in the view self-maintenance problem.

EXAMPLE 2.4.1 Consider a view V defined by

$$v(a, Y, Y, X, Z) \coloneq r(b, Y, X) \ \& \ s(Y, c, Y, Z)$$

and consider an update that consists inserting tuples $s(d, c, e, g)$, $s(e, d, e, h)$, and $s(f, c, f, i)$. In the rectified representation, we have a view $V' = \{(X, Y, Z) \mid (a, Y, Y, X, Z) \in V\}$ and V' is defined by

$$v'(X, Y, Z) \coloneq r'(X, Y) \ \& \ s'(Y, Z)$$

Note that the constants a, b, and c in the definition of v are eliminated, and variable Y is no longer repeated in the head or the second subgoal. Furthermore, the update in the rectified representation consists of inserting only $s'(f, i)$, since the other insertions do not match the subgoal $s(Y, c, Y, Z)$.

Thus, the self-maintenance problem of V' under the insertion of $s'(f, i)$, where V' is defined by $v'(X, Y, Z) \coloneq r'(X, Y) \ \& \ s'(Y, Z)$, can be studied instead of the original problem.

As we will see in the next chapter, this problem has the following solution:

- View V' is self-maintainable under the insertion of $s'(f, i)$ if and only if V' has a tuple of the form $(-, f, -)$.
- To maintain V', add all tuples (x, f, i) where V' has some tuple of the form $(x, f, -)$.

This solution can be translated back to the original problem as follows:

- View V is self-maintainable under the insertion of $s(d,c,e,g)$, $s(e,d,e,h)$, and $s(f,c,f,i)$ if and only if V has a tuple of the form $(a,f,f,-,-)$.
- To maintain V, add all tuples (a,f,f,x,i) where V has some tuple of the form $(a,f,f,x,-)$.

□

The following theorem substantiates our claim that the rectified representation is an equivalent representation for deciding view self-maintainability.

Theorem 2.4.1 *Consider a view V defined by a conjunctive query Q with no self-joins, and let V' and Q' be the corresponding view and query in the rectified representation. Consider an update U that consists of insertions and deletions to the base relations. Let U' be the corresponding update in the rectified representation. Then V is self-maintainable under U if and only if V' is self-maintainable under U'.* □

Proof:

It is easy to see that there is a one-to-one correspondence between the tuples in V and V'.

IF: Assume V' is self-maintainable under U'. To show that V is also self-maintainable under U, let $(I_1,\ldots I_n)$ and $(J_1,\ldots J_n)$ be two instances of the base relations $R_1,\ldots,R_n$ that are both consistent with V. For each $i=1,\ldots,n$, let us define $\hat{I}_i$ (resp. $\hat{J}_i$) to consist of those tuples in I_i (resp. J_i) that match subgoal G_i. It is easy to verify that $(\hat{I}_1,\ldots \hat{I}_n)$ and $(\hat{J}_1,\ldots \hat{J}_n)$ are also consistent with V.

For each i, let us now define I'_i to be the instance of relation R'_i that is the result of applying query G_i to $\hat{I}_i$. Note that there is a one-to-one correspondence between the tuples in I'_i and $\hat{I}_i$. J'_i is similarly defined. This situation is illustrated in Figure 2.4. It is easy to verify that the two instances $(I'_1,\ldots,I'_n)$ and $(J'_1,\ldots,J'_n)$ are consistent with V'. Since we assume that V' is self-maintainable under U', it follows that $Q'(U'(I'_1),\ldots,U'(I'_n))$ is identical to $Q'(U'(J'_1),\ldots,U'(J'_n))$. Since for every i, there is a one-to-one correspondence between the tuples inserted to or deleted from $\hat{I}_i$ (resp. $\hat{J}_i$) and the tuples inserted to or deleted from I'_i (resp. J'_i), we conclude that $Q(U(\hat{I}_1),\ldots,U(\hat{I}_n))$ is identical to $Q(U(\hat{J}_1),\ldots,U(\hat{J}_n))$. In other words, $Q(U(I_1),\ldots,U(I_n))$ is identical to $Q(U(J_1),\ldots,U(J_n))$, which completes the proof that V is self-maintainable under U.

ONLY-IF: Conversely, assume V is self-maintainable under U. Let $(I'_1,\ldots I'_n)$ and $(J'_1,\ldots J'_n)$ be two instances of the relations $R'_1,\ldots,R'_n$ that are consistent with V'. For each $i=1,\ldots,n$, let us define an instance I_i for relation R_i as follows: each tuple t in I_i is obtained from some tuple t' in I'_i by padding t' with the appropriate constants, if any, that occurs in subgoal G_i, and by duplicating those components in t' that correspond to those variables in G_i that are repeated. J_i is similarly defined.

Since there is a one-to-one correspondence between the tuples in I_i and I'_i on the one hand, and between the tuples in J_i and J'_i on the other hand, it is not hard to see that the two instances $(I_1,\ldots I_n)$ and $(J_1,\ldots J_n)$ are consistent with V. And since we assume that V

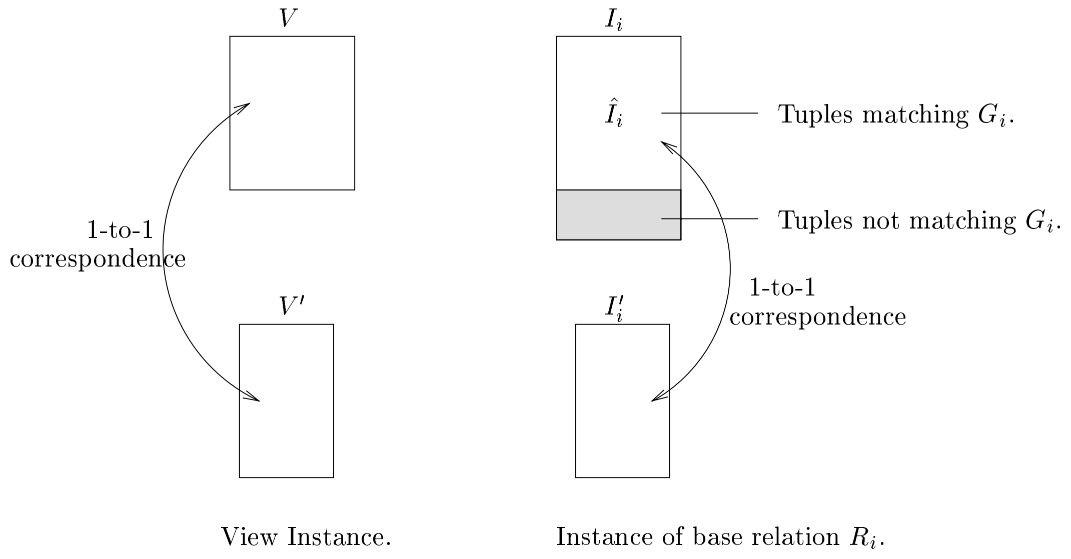


Figure 2.4: Correspondences in the rectified representation.

is self-maintainable under U, $Q(U(I_1),\ldots,U(I_n))$ must be identical to $Q(U(J_1),\ldots,U(J_n))$. For a reason similar to the one given in the *IF:* proof, it follows that $Q'(U'(I'_1),\ldots,U'(I'_n))$ is identical to $Q'(U'(J'_1),\ldots,U'(J'_n))$. Thus, V' is self-maintainable under U'. ∎

2.5 Query Containment

We will encounter a particular implication problem known in the database literature as the query containment (abbreviated QC) problem [Ull89]. Given two Datalog queries P and Q using EDB relations $E_1,\ldots,E_n$ as input, we say that P is *contained in* Q, denoted

$$P \subseteq Q,$$

if the answer to P is a subset of the answer to Q, for every instance of $E_1,\ldots,E_n$.

When the constant symbols used in the queries we want to compare represent known values, the decision of $P \subseteq Q$ will be either *true* or *false*, assuming that the containment is decidable. For instance, if the queries P and Q are defined by:

$$\begin{array}{ll} P: & a(X) \coloneq r(1,X,2) \\ Q: & a(U) \coloneq r(U,V,U) \end{array}$$

then, since 1 and 2 are not equal, we can always find an instance of R (e.g. consisting of the single tuple $(1,3,2)$) such that P returns a tuple not in the answer to Q, and the containment is false.

However, when some constant symbols used in the queries represent parameters whose value is unknown, the containment decision will be conditional, that is, will depend on the actual value of the parameters. If the containment can be decided, we can think of generating a procedure that takes these parameters as input and returns *true* or *false*. In simpler cases, we can even think of a logical expression of the parameters that can always be verified. For instance, if P and Q are defined by:

$$\begin{array}{ll} P: & a(X) \text{ :- } r(\$1, X, \$2) \\ Q: & a(U) \text{ :- } r(V, U, V) \end{array}$$

where the constant symbols \$1 and \$2 represent parameters, then the most general condition on \$1 and \$2 that guarantees $P \subseteq Q$ is simply $\$1 = \$2$.

In this thesis, we will encounter a variation of the containment problem where the instance of some of the input EDB relations is also given. In this case, the query containment is said to be *Instance-Specific*, and we call the containment problem "instance-specific query containment." When there is no confusion as to which EDB relations is given, we may omit "instance-specific." Given two queries P and Q using EDB relations $E_1, \ldots, E_n, F_1, \ldots, F_m$ as input, and given an instance of $F_1, \ldots, F_m$, we say that

$$P \subseteq_{F_1,\ldots,F_m} Q$$

if the answer to P is a subset of the answer to Q for all instances of $E_1, \ldots, E_n$. The EDB predicates f_i, whose extension is given, are called *constant predicates*. The EDB predicates e_i are called *variable predicates*.

When the extension of the constant predicates is known, we can always reformulate an instance-specific QC problem to a QC problem by eliminating any constant predicate f as follows: replace any subgoal $\neg f(\bar{X})$ with $\bigwedge_{\bar{x}}(\bar{X} \neq \bar{x})$ and any subgoal $f(\bar{X})$ with $\bigvee_{\bar{x}}(\bar{X} = \bar{x})$, where $\bar{x}$ ranges over the tuples in f's extension. Consider for example the containment between the following queries:

$$\begin{array}{ll} P: & a(X) \text{ :- } r(X, Y) \ \&\ s(Y) \\ Q: & a(U) \text{ :- } r(U, V) \ \&\ \neg t(V) \end{array}$$

where predicate r is variable, and predicates s and t are constant, say with extensions $S = \{1, 2\}$ and $T = \{2, 3\}$. The queries above can be expanded into the following queries:

$$\begin{array}{ll} P: & a(X) \text{ :- } r(X, Y) \ \&\ Y = 1 \\ & a(X) \text{ :- } r(X, Y) \ \&\ Y = 2 \\ Q: & a(U) \text{ :- } r(U, V) \ \&\ V \neq 2 \ \&\ V \neq 3 \end{array}$$

where all the input predicates are variables. Thus, we have reduced the original instance-specific containment problem to a normal containment problem.

However, there are situations where the extension of some constant predicates is not known. These constant predicates are just placeholders or parameters to the problem.

Then, it would not make much sense to eliminate these predicates. But, to parallel the case of normal query containment with parameter constants, we may want to generate a procedure thats take these constant predicates as inputs and decides containment. It may even be possible to generate a logical expression of the constant predicates that can be efficiently verified when the extensions of these predicates are available. In this thesis, we will be interested in those logical expressions that are efficient queries. Therefore, given an instance-specific query containment problem using constant predicates $f_1, \ldots, f_m$, an interesting question is whether the containment problem can be expressed as a (boolean) query over the input predicates $f_1, \ldots, f_m$. Consider for example the containment between the following queries:

$$\begin{array}{ll} P: & a(X) \text{ :- } r(X,Y) \ \&\ s(Y) \\ Q: & a(U) \text{ :- } r(U,V) \ \&\ \neg t(V) \end{array}$$

with variable predicate r and constant predicates s and t. The query containment in this example can be expressed as the following query:

$$(\forall Y)\ s(Y) \Rightarrow \neg t(Y)$$

which simply asks whether or not relations S and T are disjoint.

We will come back to this notion of instance-specific query containment and its translation to a query later in Chapter 5.

Finally, let us give a brief description of the state of the art on query containment. Containment of CQ's was first studied in [CM77] and a well known technique for testing containment is based on the notion of containment mapping. Given two CQ's P and Q, a *containment mapping* for the problem of testing $P \subseteq Q$ is a function that (1) maps variables in Q to constants in P, (2) maps constants to themselves, (3) turns Q's head to P's head, and (4) turns each subgoal in Q's body to some subgoal in P. While the problem of deciding containment of CQ's is NP-complete ([CM77]), there are special cases of CQ's for which the problem has polynomial-time solutions ([ASU79a, ASU79b, JK83, CR97]). Containment of unions of CQ's was considered in [SY80] and containment mappings can be used to test the containment. Deciding containment of unions of CQ's is Π_2^p-complete. Containment of CQ's with arithmetic comparisons was originally studied in [Klu88] and a technique based on the use of containment mappings for containment testing appears in [GU92, G*94, Gup94]. Part of our thesis will be based on this technique (see Appendix A), where query containment is expressed as a logic expression whose truth decides containment. [Mey92] recently showed deciding containment of CQ's with arithmetic comparisons to be Π_2^p-complete. Containment of unions of CQ's with negation was considered in [Sag87] and a technique based on testing a finite (but exponential) number of databases for deciding containment appears in [LS93]. While the containment for some restricted classes of nonrecursive Datalog queries with negation can be decided efficiently ([SY80]), the complexity of containment for this class in general is not known. Another dimension to the problem is to consider the presence of dependencies in deciding containment, and techniques for testing containment of CQ's based on chasing the dependencies over the queries appear in [Sag87, ASU79a, ASU79b, JK84, RSUV89, RSUV93]. Finally, to the best of our knowledge, the complexity of deciding instance-specific query containment and the problem of whether instance-specific query containment is expressible as a query (in terms of the constant predicates) have not been studied in the literature.

Chapter 3

Strict View Self-Maintenance

We start with the strict view self-maintenance problem, i.e., the view self-maintenance problem where:

- No auxiliary views are used,
- No base relations are used, and
- No dependencies on the base relations are used.

In this chapter, we consider the problem of maintaining views

- Defined by conjunctive queries with no self-joins,
- Under updates to a single base relation.

We will show that for this simple case, strict view self-maintenance admits solutions in simple query forms.

Throughout this chapter, the view to maintain is defined by a conjunctive query Q represented as follows:

$$v(\bar{X}', \bar{Y}', \bar{Z}') :- r(\bar{X}, \bar{Y}) \ \& \ S(\bar{Y}, \bar{Z}) \tag{3.1}$$

where $\bar{X}$, $\bar{Y}$ and $\bar{Z}$ denote disjoint sets of variables, $\bar{X}'$, $\bar{Y}'$ and $\bar{Z}'$ denote subsets of $\bar{X}$, $\bar{Y}$ and $\bar{Z}$ respectively, r is the predicate for the updated relation, and S denotes a conjunction of subgoals whose relations are not updated.

Let us define the terminology used:

- The variables in $\bar{X}$ and $\bar{Y}$ are called the *updated* variables.
- The variables in $\bar{Y}$ are called the *join* variables.
- The variables in $\bar{Z}$ are called the *private* variables.
- A variable from $\bar{X}$, $\bar{Y}$, or $\bar{Z}$ is said to be *exposed* if it appears in the head of Q, i.e., if it is in $\bar{X}'$, $\bar{Y}'$, or $\bar{Z}'$. It is *hidden* otherwise.

Section 3.1, *Deletions,* presents solutions to the view self-maintenance problem for views defined by conjunctive queries without self-joins, under single deletions and multiple deletions from a single relation.

Section 3.2, *Insertions,* solves the view self-maintenance problem under single insertions and multiple insertions into a single relation.

Section 3.3, *Mixing Insertions with Deletions,* considers the view self-maintenance problem under both insertions and deletions to a single relation.

Section 3.4, *Summary,* summarizes the results in this chapter and points out difficulties when considering self-joins in the view definitions or updates across more than one base relation.

3.1 Deletions

We consider deletions first since they are simpler to deal with than insertions. Insertions will be handled in the next section.

Referring to the generic view definition (3.1), we consider the following three nonoverlapping cases successively:

- All updated variables are exposed: $\bar{X}' = \bar{X}$ and $\bar{Y}' = \bar{Y}$.
- Some updated variables are hidden, but all join variables are exposed: $\bar{X}' \subset \bar{X}$ and $\bar{Y}' = \bar{Y}$.
- Some join variables are hidden: $\bar{X}' \subseteq \bar{X}$ and $\bar{Y}' \subset \bar{Y}$.

3.1.1 All Updated Variables Exposed

Theorem 3.1.1 *Let view V be defined by $v(\bar{X}, \bar{Y}, \bar{Z}')$:- $r(\bar{X}, \bar{Y})$ & $S(\bar{Y}, \bar{Z})$, where $\bar{Z}' \subseteq \bar{Z}$. V is always self-maintainable under the deletion of $r(\bar{a}, \bar{b})$. To maintain it, delete all tuples of the form $V(\bar{a}, \bar{b}, -)$.* □

Proof: The tuples in V that only depend on $r(\bar{a}, \bar{b})$ are exactly those of the form $V(\bar{a}, \bar{b}, -)$, since all the updated variables are exposed. ■

3.1.2 Some updated variables hidden but all join variables exposed

Referring to the generic view definition (3.1), this subsection deals with the case where $\bar{X}' \subset \bar{X}$ and $\bar{Y}' = \bar{Y}$. To emphasize the fact that $\bar{X}'$ is a strict subset of $\bar{X}$ in this subsection, we will use $\bar{X}''$ instead of $\bar{X}'$. This notation serves no other purposes than a syntactic reminder that there are variables in $\bar{X}$ that are not in $\bar{X}''$.

Theorem 3.1.2 *Let view V be defined by $v(\bar{X}'', \bar{Y}, \bar{Z}')$:- $r(\bar{X}, \bar{Y})$ & $S(\bar{Y}, \bar{Z})$, where $\bar{X}'' \subset \bar{X}$ and $\bar{Z}' \subseteq \bar{Z}$. V is self-maintainable under the deletion of $r(\bar{a}, \bar{b})$ if and only if it has no tuples of the form $v(\bar{a}'', \bar{b}, -)$. We use $\bar{a}''$ to denote the $\bar{X}''$ components of $\bar{a}$. Furthermore, in this situation, V is not affected by the deletion of $r(\bar{a}, \bar{b})$.* □

Proof:

IF: If there is no $v(\bar{a}'', \bar{b}, -)$, then no V-tuples depend on $r(\bar{a}, \bar{b})$. So the deletion of $r(\bar{a}, \bar{b})$ does not affect V.

ONLY-IF: Assume there is some $v(\bar{a}'', \bar{b}, -)$. We will construct two databases D_1 and D_2 that are both consistent with V before the deletion, but that derive differently [1] after the deletion.

First, D_1 is taken to be the canonical database, which is constructed from V in the usual way. We already know that D_1 is consistent with V. Furthermore, since the subgoal $r(\bar{X}, \bar{Y})$ has at least one hidden X-variable, D_1 cannot contain $r(\bar{a}, \bar{b})$. Thus, the deletion of $r(\bar{a}, \bar{b})$ has no effect on V. Furthermore, as will be clear in a moment, even if we include $r(\bar{a}, \bar{b})$ in D_1, the resulting database is still consistent with V, and the deletion still has no effect on V. Thus, knowing that $r(\bar{a}, \bar{b})$ is in the underlying database will not affect the completeness of our result.

To construct D_2, it is useful to layout the tuples in D_1 as shown in Table 3.1. In this layout, V is partitioned into V_1 and V_2, where V_1 contains all the V-tuples that agree with $\bar{a}$ over $\bar{X}''$ and with $\bar{b}$ over $\bar{Y}$, and V_2 contains those tuples $(\bar{x}'', \bar{y}, \bar{z}')$ such that $(\bar{x}'', \bar{y}) \neq (\bar{a}'', \bar{b})$. For each tuple $(\bar{x}'', \bar{y}, \bar{z}')$ in V, we construct the corresponding tuple $(\bar{x}''+, \bar{y})$ in R by extending $\bar{x}''$ to the remaining hidden X-variables with new symbols, and the corresponding tuples $(\bar{y}, \bar{z}'*)$ in S by extending $\bar{z}'$ to the remaining hidden Z-variables with new symbols. The new symbols are created for each new line in the table. Also, we use "$*$" to denote zero or more new symbols and "+" to denote one or more new symbols. Thus, database D_1 consists of $M \cup O \cup N \cup P$.

Database D_2 is obtained from D_1 by substituting $M' = \{r(\bar{a}, \bar{b})\}$ for M. Since M and O are disjoint because $(\bar{x}'', \bar{y}) \neq (\bar{a}'', \bar{b})$, the substitution does not affect O. In other words, $D_2 = M' \cup O \cup N \cup P$, as depicted in Table 3.1.

The following about database D_2 holds:

- $Q(D_2) \supseteq V$:
  - As in D_1, V_2 can be derived from O and P.
  - Each line in V_1 can be derived by joining M' (consisting of $r(\bar{a}, \bar{b})$) with the corresponding line in N over $\bar{Y}$ (whose components have value $\bar{b}$).
- $Q(D_2) \subseteq V$:
  - Suppose $r(\bar{a}, \bar{b})$ joins with some set T' of tuples from $N \cup P$ to derive some tuple t that is not in V. Since the join is over $\bar{Y}$, then in D_1, any tuple from M would have joined with T' to derive t, which is impossible since $Q(D_1) \subseteq V$.

[1]Recall from Section 1.3 the phrase "derive differently" used here to mean that after the deletion, D_2 derives, through the query defining V, a view that is different from what D_1 derives after the deletion.

$v(\bar{X}'', \bar{Y}, \bar{Z}')$		$r(\bar{X}, \bar{Y})$		$S(\bar{Y}, \bar{Z})$	
	$\vdots$		$\vdots$		$\vdots$
V_1	$\bar{a}'', \bar{b}, \bar{z}'$	M	$\bar{a}''+, \bar{b}$	N	$\bar{b}, \bar{z}'*$
	$\vdots$		$\vdots$		$\vdots$
	$\vdots$		$\vdots$		$\vdots$
V_2	$\bar{x}'', \bar{y}, \bar{z}'$	O	$\bar{x}''+, \bar{y}$	P	$\bar{y}, \bar{z}'*$
	$\vdots$		$\vdots$		$\vdots$

Canonical database $D_1 = M \cup O \cup N \cup P$.

$v(\bar{X}'', \bar{Y}, \bar{Z}')$		$r(\bar{X}, \bar{Y})$		$S(\bar{Y}, \bar{Z})$	
	$\vdots$				$\vdots$
V_1	$\bar{a}'', \bar{b}, \bar{z}'$	M'	$\bar{a}, \bar{b}$	N	$\bar{b}, \bar{z}'*$
	$\vdots$				$\vdots$
	$\vdots$		$\vdots$		$\vdots$
V_2	$\bar{x}'', \bar{y}, \bar{z}'$	O	$\bar{x}''+, \bar{y}$	P	$\bar{y}, \bar{z}'*$
	$\vdots$		$\vdots$		$\vdots$

Database instance $D_2 = M' \cup O \cup N \cup P$.

Table 3.1: Counterexample in the proof of Theorem 3.1.2.

  - As in D_1, when joined with $N \cup P$, O can only derive tuples in V.

- V_1 is not supported by O: A tuple in O can only derive a V-tuple whose $\bar{X}''$ component is not $\bar{a}''$ or whose $\bar{Y}$ component is not $\bar{b}$. Thus the derived V-tuple cannot be in V_1.

So D_2 is consistent with V and the deletion of $r(\bar{a}, \bar{b})$ from D_2 causes V to loose all its V_1-tuples. Thus $Q(D_2 - r(\bar{a}, \bar{b})) = V_2$, $Q(D_1 - r(\bar{a}, \bar{b})) = V$, and $V_2 \subset V$. In other words, $Q(D_1 - r(\bar{a}, \bar{b})) \neq Q(D_2 - r(\bar{a}, \bar{b}))$. ∎

EXAMPLE 3.1.1 Consider the view definition

$$v(U, Y, W, Z) \text{ :- } r(X, U, Y, W) \;\&\; p_1(U, Y) \;\&\; p_2(Y, Z) \;\&\; p_3(W, Z) \;\&\; p_4(T)$$

and consider the deletion of $r(x, a, b, c)$. In this view definition, the join variables U, Y, W are all exposed, but the updated variable X is hidden. Applying Theorem 3.1.2, a given instance V is self-maintainable under the deletion if and only if V has no tuples of the form

$v(U,Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
a,b,c,z'	x_1,a,b,c	a,b	b,z'	c,z'	t_1
a,b,c,z''	x_2,a,b,c	$-$	b,z''	c,z''	t_2
a',b,c,z'	x_3,a',b,c	a',b	$-$	$-$	t_3
a',b,c,z''	x_4,a',b,c	$-$	$-$	$-$	t_4
Nothing deleted.	Delete x,a,b,c				

Canonical database D_1.

$v(U,Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
a,b,c,z'	x,a,b,c	a,b	b,z'	c,z'	t_1
a,b,c,z''	$-$	$-$	b,z''	c,z''	t_2
a',b,c,z'	x_3,a',b,c	a',b	$-$	$-$	t_3
a',b,c,z''	x_4,a',b,c	$-$	$-$	$-$	t_4
a,b,c,z' and a,b,c,z'' deleted.	Delete x,a,b,c				

Database instance D_2 derives differently from D_1 after the deletion.

Figure 3.1: A non-self-maintainable view instance from Example 3.1.1.

$(a,b,c,-)$. Furthermore, if V satisfies this condition, then the deletion does not affect the view.

In fact, any view instance V that has no $(a,b,c,-)$ tuples cannot possibly have any tuple that depends on $r(x,a,b,c)$. Thus, V is not affected by the deletion of $r(x,a,b,c)$. Conversely, consider an instance of V that contains some $(a,b,c,-)$ tuples, as shown in Figure 3.1, where the first table shows a database instance D_1 and the view, before and after the deletion, and the second table shows another database instance D_2 and the view, before and after the deletion.

The base instances D_1 and D_2 are a counterexample showing that the view instance is not self-maintainable under the deletion of $r(x,a,b,c)$. Even though there may be other counterexamples, we choose this one to follow the general construction method presented in the proof and shown in Table 3.1. □

3.1.3 Some join variables hidden

Referring to the generic view definition (3.1), this subsection deals with the case where $\bar{Y}' \subset \bar{Y}$. To emphasize the fact that $\bar{Y}'$ is a strict subset of $\bar{Y}$ in this subsection, we will use $\bar{Y}''$ instead of $\bar{Y}'$.

Theorem 3.1.3 *Let view V be defined by $v(\bar{X}', \bar{Y}'', \bar{Z}')$:- $r(\bar{X}, \bar{Y})$ & $S(\bar{Y}, \bar{Z})$, where $\bar{X}' \subseteq \bar{X}$, $\bar{Y}'' \subset \bar{Y}$, and $\bar{Z}' \subseteq \bar{Z}$. V is self-maintainable under the deletion of $r(\bar{a}, \bar{b})$ if and only if it has no tuples of the form $v(\bar{a}', \bar{b}'', -)$. We use $\bar{a}'$ to denote the $\bar{X}'$ components of $\bar{a}$, and $\bar{b}''$ the $\bar{Y}''$ components of $\bar{b}$. Furthermore, in this situation, V is not affected by the deletion of $r(\bar{a}, \bar{b})$.* □

Proof:

IF: If there is no $v(\bar{a}', \bar{b}'', -)$, then no V-tuples depend on $r(\bar{a}, \bar{b})$. So the deletion of $r(\bar{a}, \bar{b})$ does not affect V.

ONLY-IF: Assume V contains tuple $(\bar{a}', \bar{b}'', \bar{c}')$, for some $\bar{c}'$. We will construct a counterexample with two databases D_1 and D_2 that are both consistent with V before the deletion but that derive differently after the deletion.

Again, D_1 is taken to be the canonical database and D_1 is consistent with V. Furthermore, since the subgoal $r(\bar{X}, \bar{Y})$ has at least one hidden Y-variable, D_1 cannot contain $r(\bar{a}, \bar{b})$. Thus the deletion of $r(\bar{a}, \bar{b})$ has no effect on V. Furthermore, as will be clear in a moment, even if we include $r(\bar{a}, \bar{b})$ in D_1, the resulting database is still consistent with V, and the deletion still has no effect on V. Thus, knowing that $r(\bar{a}, \bar{b})$ is in the underlying database will not affect the completeness of our result.

Before we present the particular layout for D_1 that will be used to construct D_2, we need to partition the subgoals $S(\bar{Y}, \bar{Z})$ into $S_1(\bar{Y}_1, \bar{Z}_1)$ and $S_2(\bar{Y}_2, \bar{Z}_2)$, defined as follows:

- Initially S_1 contains all subgoals in S that have a hidden Y-variable. There is at least one such subgoal.
- Any remaining subgoal in S that shares a hidden Z-variable with some subgoal currently in S_1 will be assigned to S_1.
- Any subgoal in S that remains unassigned is assigned to S_2.

We observe the following properties about S_1 and S_2:

- All hidden Y-variables are in S_1, i.e., $\bar{Y}_1 \supseteq \bar{Y} - \bar{Y}''$.
- All Y-variables in S_2 are exposed, i.e., $\bar{Y}_2 \subseteq \bar{Y}''$.
- No hidden Z-variables are shared between S_1 and S_2, i.e., $(\bar{Z}_1 - \bar{Z}') \cap (\bar{Z}_2 - \bar{Z}') = \emptyset$.

Figure 3.2 shows how variables are shared among the subgoals.

To construct D_2, we arrange the tuples in D_1 as shown in Table 3.2. In this layout, V is partitioned into V_1 and V_2, where V_1 contains all tuples $v(\bar{a}', \bar{b}'', \bar{z}')$ that agree with $\bar{c}'$ over $\bar{Z}_1$, and V_2 contains those tuples $(\bar{x}', \bar{y}'', \bar{z}')$ such that either $\bar{x}' \neq \bar{a}'$, $\bar{y}'' \neq \bar{b}''$ or $\bar{z}'$ does not agree with $\bar{c}'$ over $\bar{Z}_1$. Note that each line in N agrees with $\bar{b}''$ over $\bar{Y}_2$ and with $\bar{c}'_1$ over $\bar{Z}_1 \cap \bar{Z}_2$. Database D_1 thus consists of $M \cup O \cup N \cup P$.

D_2 is obtained from D_1 by substituting $M' = \{r(\bar{a}, \bar{b}), S_1(\bar{b}_1, \bar{c}'_1*)\}$ for M. This substitution does not affect O since M and O are disjoint because the hidden variables on different

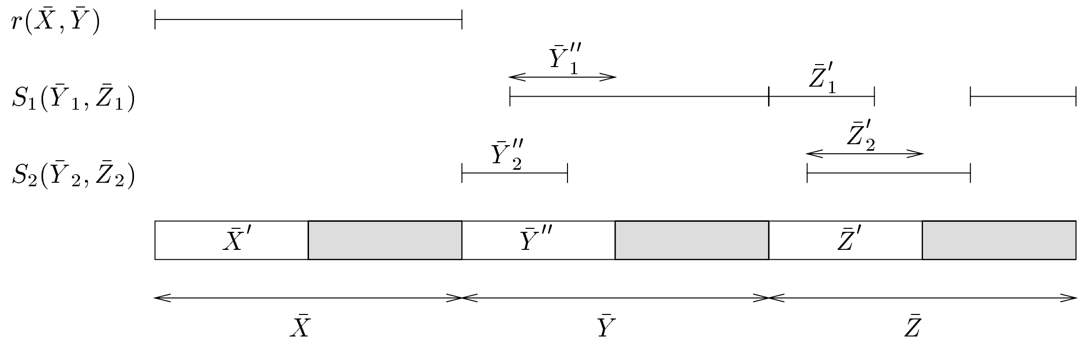


Figure 3.2: Subgoals and variables used in a view definition with hidden join variables.

lines are assigned different values. In other words, $D_2 = M' \cup O \cup N \cup P$, as depicted in Table 3.2.

Database D_2 has the following properties:

- $Q(D_2) \supseteq V$:
  - As in D_1, V_2 can be derived from O and P
  - Each line in V_1 can be derived by joining M' with the corresponding line in N over $\bar{Y}_1 \cap \bar{Y}_2$ (whose components all agree with $\bar{b}''$) and over $\bar{Z}_1 \cap \bar{Z}_2$ (whose components all agree with $\bar{c}'$). See Figure 3.2.
- $Q(D_2) \subseteq V$:
  - First, note that $r(\bar{a},\bar{b})$ can only join with the S_1-tuples from M'. Now, suppose M' joins with some set T' of tuples from $N \cup P$ to derive some tuple t that is not in V. Since the join is over a subset of $\bar{Y}''$ $(\bar{Y}_1 \cap \bar{Y}_2)$ and on a subset of $\bar{Z}_1$ $(\bar{Z}_1 \cap \bar{Z}_2)$, then any line from M would have joined with T' to derive t, which is impossible since $Q(D_1) \subseteq V$.
  - Also, any tuple $\bar{x}'*, \bar{y}''+$ from O can only join with the S_1-tuples on the same line and thus can only derive tuples in V.
- O cannot contribute to V_1: As mentioned above, any tuple $\bar{x}'*, \bar{y}''+$ from O can only join with the S_1-tuples on the same line. So any tuple $v(\bar{x}',\bar{y}'',\bar{z}')$ that O can derive is not in V_1, since we already know either $\bar{x}' \neq \bar{a}'$, $\bar{y}'' \neq \bar{b}''$, or $\bar{z}'$ disagrees with $\bar{c}'$ over $\bar{Z}_1$.

So D_2 is consistent with V, and the deletion of $r(\bar{a},\bar{b})$ causes V to loose all its V_1-tuples. In other words, $Q(D_2 - r(\bar{a},\bar{b}))$ has strictly fewer tuples than $Q(D_1 - r(\bar{a},\bar{b}))$. ∎

$v(\bar{X}', \bar{Y}'', \bar{Z}')$		$r(\bar{X}, \bar{Y})$		$S_1(\bar{Y}_1, \bar{Z}_1)$	$S_2(\bar{Y}_2, \bar{Z}_2)$	
	$\vdots$		$\vdots$	$\vdots$		$\vdots$
V_1	$\bar{a}', \bar{b}'', \bar{z}'$	M	$\bar{a}'*, \bar{b}''+$	$\bar{b}_1''+, \bar{c}_1'*$	N	$\bar{b}_2'', \bar{z}_2'*$
	$\vdots$		$\vdots$	$\vdots$		$\vdots$
	$\vdots$		$\vdots$	$\vdots$		$\vdots$
V_2	$\bar{x}', \bar{y}'', \bar{z}'$	O	$\bar{x}'*, \bar{y}''+$	$\bar{y}_1''+, \bar{z}_1'*$	P	$\bar{y}_2'', \bar{z}_2'*$
	$\vdots$		$\vdots$	$\vdots$		$\vdots$

Canonical database $D_1 = M \cup O \cup N \cup P$.

$v(\bar{X}', \bar{Y}'', \bar{Z}')$		$r(\bar{X}, \bar{Y})$		$S_1(\bar{Y}_1, \bar{Z}_1)$	$S_2(\bar{Y}_2, \bar{Z}_2)$	
	$\vdots$					$\vdots$
V_1	$\bar{a}', \bar{b}'', \bar{z}'$	M'	$\bar{a}, \bar{b}$	$\bar{b}_1, \bar{c}_1'*$	N	$\bar{b}_2'', \bar{z}_2'*$
	$\vdots$					$\vdots$
	$\vdots$		$\vdots$	$\vdots$		$\vdots$
V_2	$\bar{x}', \bar{y}'', \bar{z}'$	O	$\bar{x}'*, \bar{y}''+$	$\bar{y}_1''+, \bar{z}_1'*$	P	$\bar{y}_2'', \bar{z}_2'*$
	$\vdots$		$\vdots$	$\vdots$		$\vdots$

Database instance $D_2 = M' \cup O \cup N \cup P$.

Table 3.2: Counterexample in the proof of Theorem 3.1.3.

EXAMPLE 3.1.2 Consider the view definition

$$v(Y, W, Z) :- r(X, U, Y, W) \ \& \ p_1(U, Y) \ \& \ p_2(Y, Z) \ \& \ p_3(W, Z) \ \& \ p_4(T)$$

and consider the deletion of $r(x, a, b, c)$. In this view definition, the join variable U is hidden. Applying Theorem 3.1.3, a given instance V is self-maintainable under the deletion if and only if V has no tuples of the form $(b, c, -)$. Furthermore, if V satisfies this condition, then the deletion does not affect the view. In fact, consider an instance of V that contains some $(b, c, -)$ tuples, as shown in Figure 3.3, where the first table shows a database instance D_1 and the view, before and after the deletion, and the second table shows another database instance D_2 and the view, before and after the deletion.

The base instances D_1 and D_2 are a counterexample showing that the view instance is not self-maintainable under the deletion of $r(x, a, b, c)$. Even though there may be other counterexamples, we choose this one to follow the general construction method presented in the proof and shown in Table 3.2. $\square$

$v(Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
b,c,z'	x_1,a_1,b,c	a_1,b	b,z'	c,z'	t_1
b,c,z''	x_2,a_2,b,c	a_2,b	b,z''	c,z''	t_2
b,c',z'''	x_3,a_3,b,c'	a_3,b	b,z'''	c',z'''	t_3
Nothing deleted.	Delete x,a,b,c				

Canonical database D_1.

$v(U,Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
b,c,z'	x,a,b,c	a,b	b,z'	c,z'	t_1
b,c,z''	$-$	$-$	b,z''	c,z''	t_2
b,c',z'''	x_3,a_3,b,c'	a_3,b	b,z'''	c',z'''	t_3
b,c,z' and b,c,z'' deleted.	Delete x,a,b,c				

Database instance D_2 derives differently from D_1 after the deletion.

Figure 3.3: A non-self-maintainable view instance from Example 3.1.2.

3.1.4 Multiple Deletions

All Updated Variables Exposed

Theorem 3.1.4 *Let view V be defined by $v(\bar{X},\bar{Y},\bar{Z}')$:- $r(\bar{X},\bar{Y})$ & $S(\bar{Y},\bar{Z})$, where $\bar{Z}' \subseteq \bar{Z}$. Let U be an update that consists of the deletion of $r(\bar{x}_1,\bar{y}_1),\ldots,r(\bar{x}_n,\bar{y}_n)$. V is always self-maintainable under U. To maintain it, delete all tuples of the form $V(\bar{x}_i,\bar{y}_i,-)$, for for $i=1,\ldots,n$.* □

Proof: Any V-tuple that depends on tuple $r(\bar{x}_i,\bar{y}_i)$ must be of the form $(\bar{x}_i,\bar{y}_i,-)$. Conversely, any V-tuple of the form $(\bar{x}_i,\bar{y}_i,-)$ not only depends on $r(\bar{x}_i,\bar{y}_i)$, but also cannot depend on r-tuples other that $r(\bar{x}_i,\bar{y}_i)$. ■

Some Updated Variables Hidden

Theorem 3.1.5 *Let view V be defined by $v(\bar{X}',\bar{Y}',\bar{Z}')$:- $r(\bar{X},\bar{Y})$ & $S(\bar{Y},\bar{Z})$, where either $\bar{X}' \subset \bar{X}$, $\bar{Y}' \subset \bar{Y}$, or both. Let U be an update that consists of the deletion of $r(\bar{x}_1,\bar{y}_1),\ldots,r(\bar{x}_n,\bar{y}_n)$. V is self-maintainable under U if and only if for every $i=1,\ldots,n$, V has no tuples of the form $v(\bar{x}_i',\bar{y}_i',-)$. Furthermore, in this situation, V is not affected by U.* □

Proof:

IF: No V-tuples depend on any $r(\bar{x}_i,\bar{y}_i)$. Thus, removal of all the $r(\bar{x}_i,\bar{y}_i)$ does not affect V.

ONLY-IF: Proof by constructing counterexamples very similar to the case of single deletions. ∎

3.2 Insertions

In the previous section, the problem of view self-maintenance under deletions has a solution that is simple to characterize: for views defined by $v(\bar{X}', \bar{Y}', \bar{Z}') \text{ :- } r(\bar{X}, \bar{Y})\ \&\ S(\bar{Y}, \bar{Z})$ and under the deletion of $r(\bar{a}, \bar{b})$, the solution can be expressed independently of how the variables are distributed among the subgoals in $S(\bar{Y}, \bar{Z})$.

By contrast, as will be clearer in a moment, the solution to the problem under insertions cannot be expressed independently of the internal structure of $S(\bar{Y}, \bar{Z})$. The following example illustrates this point.

EXAMPLE 3.2.1 Consider the following two different view definitions:

$$\begin{array}{ll} Q_1: & v(X,Y) \text{ :- } r(X,Y)\ \&\ t(X,Y) \\ Q_2: & v(X,Y) \text{ :- } r(X,Y)\ \&\ t_1(X)\ \&\ t_2(Y). \end{array}$$

Consider the insertion of $r(a,b)$ and the problem of maintaining a view V defined by either Q_1 or Q_2. One can easily verify that while V is self-maintainable under the insertion if and only if $V(a,b)$ holds in the first case, this condition is no longer necessary for self-maintainability of V in the second case. In fact, it is not difficult to verify that in the second case, a necessary and sufficient condition for V's self-maintainability under the insertion is that $V(a,-) \wedge V(-,b)$ holds. The latter condition strictly subsumes the former. □

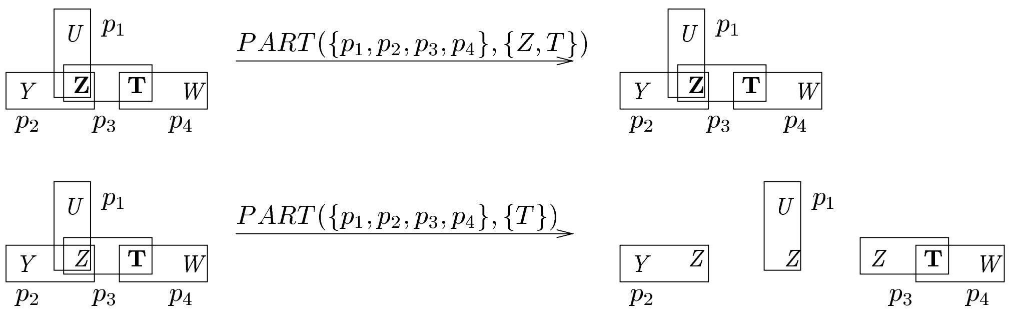
This example suggests that in order to express the solution to the self-maintenance problem under insertions, we need some way to characterize syntactically the internal structure of $S(\bar{Y}, \bar{Z})$.

In this section, we first define the concept of *subgoal partitioning*, a concept that is adequate to capture the structure of the view definition for the purpose solving the self-maintenance problem. We then use the concept to present the solution for the special case where all variables are exposed. This solution extends to the general case where all join variables are exposed. The case where some join variables hidden is considered next. Finally, we consider the problem of self-maintenance under multiple insertions into a single base relation.

3.2.1 Subgoal Partitioning

The concept of subgoal partitioning will be applied to organize into groups the subgoals in the body of a view definition with nonupdated relations. But, regardless of the use context, the concept can be defined as follows.

Definition 3.2.1 (**Subgoal Partitioning**) Let $S(\bar{U})$ be a set of subgoals with distinct predicates, where $\bar{U}$ represents the set of variables used in the subgoals. Let $\bar{V}$ be a subset



U
p_1
Y
$\mathbf{Z}$
$\mathbf{T}$
W
p_2
p_3
p_4
$PART(\{p_1, p_2, p_3, p_4\}, \{Z, T\})$
$PART(\{p_1, p_2, p_3, p_4\}, \{T\})$
Z

Figure 3.4: Examples of subgoal partitioning.

of $\bar{U}$. We define $PART(S(\bar{U}), \bar{V})$ to be the finest partition of $S(\bar{U})$ into groups $S_1(\bar{U}_1)$, $S_2(\bar{U}_2)$, ..., such that no two groups share some $\bar{V}$-variables. □

We can equivalently define $PART(S(\bar{U}), \bar{V})$ as follows. Consider a graph whose nodes correspond to the subgoals in $S(\bar{U})$ and where two nodes are connected if the corresponding subgoals share some $\bar{V}$-variable. Then the connected components in the graph correspond to the groups $S_1(\bar{U}_1)$, $S_2(\bar{U}_2)$, ...

EXAMPLE 3.2.2 Consider the set S of subgoals:

$$p_1(U, Z), p_2(Y, Z), p_3(Z, T), p_4(W, T)$$

$PART(S, \{Z, T\})$ consists of a single group that includes all the subgoals, since a group that contains $p_3(Z, T)$ shares either Z or T with any other subgoal. By contrast, $PART(S, \{T\})$ consists of the three groups $\{p_1(U, Z)\}$, $\{p_2(Y, Z)\}$, and $\{p_3(Z, T), p_4(W, T)\}$. Figure 3.4 illustrates these two cases in a connection hypergraph notation. In Figure 3.4, the variables in the second parameter of $PART(., .)$ are shown in boldface. These variables behave like glue that holds the subgoals together while we are trying to split them apart. □

Algorithm for computing $PART(S(\bar{U}), \bar{V})$

There is a simple one-pass algorithm that computes the groups in $PART(S(\bar{U}), \bar{V})$. Scan the given list S of subgoals and consider each subgoal in turn. If the subgoal has no $\bar{V}$-variable, assign it to a new group. Otherwise, look for an existing group that shares some $\bar{V}$-variable with the subgoal. If none can be found, assign the subgoal to a new group. Otherwise, merge all such groups and assign the subgoal to the result.

Partitioning the subgoals that use nonupdated relations

Consider a view V defined by

$$v(\bar{X}', \bar{Y}', \bar{Z}') \text{ :- } r(\bar{X}, \bar{Y}) \ \& \ S(\bar{Y}, \bar{Z})$$

where r is the updated relation and $S(\bar{Y}, \bar{Z})$ represents the subgoals that use nonupdated relations. We are mainly interested in partitioning the latter subgoals as follows:

$$PART(S(\bar{Y}, \bar{Z}), \bar{Z}).$$

Let us first introduce some notation we will be using to describe the groups in the partition. $PART(S(\bar{Y}, \bar{Z}), \bar{Z})$ partitions $S(\bar{Y}, \bar{Z})$ into groups of subgoals. We use g to denote a particular group, $\bar{Y}_g$ to denote the $\bar{Y}$-variables used in group g, and $\bar{Z}_g$ to denote the $\bar{Z}$-variables used in group g. The set of subgoals in group g is written as $S_g(\bar{Y}_g, \bar{Z}_g)$.

The following properties of $PART(S(\bar{Y}, \bar{Z}), \bar{Z})$ may be obvious to the reader. We emphasize them here because we will be using them in showing the results we present later.

- No two groups share any $\bar{Z}$-variable. Consequently, given a constant $\bar{b}$ (of the same arity as $\bar{Y}$), we can always decompose the query

  $$\{\bar{z} : S(\bar{b}, \bar{z})\}$$

  into *independent* queries, one for each group g, as follows:

  $$\{\bar{z}_g : S_g(\bar{b}_g, \bar{z}_g)\}$$

  where $\bar{b}_g$ denotes the $\bar{Y}_g$ components of $\bar{b}$, and $\bar{z}_g$ denotes the $\bar{Z}_g$ components of $\bar{z}$.

- Any group having no $\bar{Z}$-variables consists of a single subgoal.

- Any group having some $\bar{Z}$-variables consists of subgoals that are all "connected" by $\bar{Z}$-variables. Consequently, suppose a database instance contains a set T of tuples, one tuple for each nonupdated relation , that satisfies $S(\bar{Y}, \bar{Z})$. Suppose the $\bar{Z}$ components in T's tuples are unique, i.e., they do not appear anywhere else in the database instance. Then if the satisfaction of $S(\bar{Y}, \bar{Z})$ involves some tuple from T, it will involve all tuples from T.

Referring to the generic view definition (3.1), the next three subsections consider the following three nonoverlapping cases respectively:

- All join variables are exposed: $\bar{Y}' = \bar{Y}$.

- No Group has both Hidden Join and Exposed Private Variables: $\bar{Y}' \subset \bar{Y}$ and for every group g in $PART(S(\bar{Y}, \bar{Z}), \bar{Z})$, either $\bar{Y}_g \subseteq \bar{Y}'$ or $\bar{Z}_g \cap \bar{Z}' = \emptyset$.

- Some Group has both Hidden Join and Exposed Private Variables: $\bar{Y}' \subset \bar{Y}$ and there is a group g in $PART(S(\bar{Y}, \bar{Z}), \bar{Z})$ such that either $\bar{Y}_g \not\subseteq \bar{Y}'$ and $\bar{Z}_g \cap \bar{Z}' \neq \emptyset$.

3.2.2 All Join Variables Exposed

Theorem 3.2.1 *Let view V be defined by $v(\bar{X}', \bar{Y}, \bar{Z}')$:- $r(\bar{X}, \bar{Y}), S(\bar{Y}, \bar{Z})$, where $\bar{X}' \subseteq \bar{X}$ and $\bar{Z}' \subseteq \bar{Z}$. V is self-maintainable under the insertion of $r(\bar{a}, \bar{b})$ if and only if the following holds:*

$$\bigwedge_{g \in PART(S(\bar{Y},\bar{Z}),\bar{Z})} (\exists \bar{Y}) V(-, \bar{Y}, -) \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \tag{3.2}$$

To maintain V (when it is self-maintainable), insert all tuples $(\bar{a}', \bar{b}, \bar{z}')$ where $\bar{z}'_g$, the $\bar{Z}_g$ components of $\bar{z}'$, is obtained from the query

$$\{\bar{z}'_g \mid (\exists \bar{Y}, \bar{Z}') V(-, \bar{Y}, \bar{Z}') \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \wedge \bar{Z}' =_{\bar{Z}_g} \bar{z}'_g\}$$

□

Proof:

Let us use $\bigwedge_g \alpha_g$ as a shorthand for condition (3.2).

IF: Assume that condition (3.2) holds. Let D be an arbitrary database instance consistent with V. We need to show that $Q(D \cup r(\bar{a}, \bar{b}))$ is independent of D.

$$\begin{aligned} & Q(D \cup r(\bar{a}, \bar{b})) \\ & = Q(D) \cup \{(\bar{a}', \bar{b}, \bar{z}') \mid S(\bar{b}, \bar{z}') \in D\} \\ & = V \cup \{(\bar{a}', \bar{b})\} \times \{(\bar{z}') \mid \textstyle\bigwedge_{g \in PART(S(\bar{Y},\bar{Z}),\bar{Z})} S_g(\bar{b}_g, \bar{z}_g) \in D\} \end{aligned}$$

For each group g, we now show that $\{\bar{z}'_g \mid S_g(\bar{b}_g, \bar{z}_g) \in D\}$ does not depend on D. The key is to show the following containment:

$$\{\bar{z}'_g \mid S_g(\bar{b}_g, \bar{z}_g) \in D\} \subseteq \{\bar{z}'_g \mid (\exists \bar{Y}, \bar{Z}') V(-, \bar{Y}, \bar{Z}') \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \wedge \bar{Z}' =_{\bar{Z}_g} \bar{z}'_g\}$$

Since α_g holds, V has some tuple $(\bar{x}', \bar{y}, \bar{t}')$, such that $\bar{y} =_{\bar{Y}_g} \bar{b}$. Since D is consistent with V, D must contain some set T of tuples $r(\bar{x}'*, \bar{y})$ and $S_h(\bar{y}_h, \bar{t}'_h*)$, where $S_h(\bar{Y}_h, \bar{Z}_h)$ denotes the set of subgoals in S other than those in S_g, $\bar{x}'*$ denotes some extension of $\bar{x}'$ to the hidden components in $\bar{X}$, and $\bar{t}'_h*$ denotes some extension of $\bar{t}'_h$ to the hidden components in $\bar{Z}_h$. Then, any tuples $S_g(\bar{b}_g, \bar{z}'_g*)$ would have joined with T to derive a V-tuple that agrees with both $\bar{b}$ over $\bar{Y}_g$ and $\bar{z}'_g$ over $\bar{Z}'_g$, as depicted in Table 3.3.

Conversely, any V-tuple $(\bar{x}', \bar{y}, \bar{z}'_g*)$ such that $\bar{y} =_{\bar{Y}_g} \bar{b}$ implies the presence in D of some tuples $S_g(\bar{b}_g, \bar{z}'_g*)$. In other words, the inverse containment also holds:

$$\{\bar{z}'_g \mid (\exists \bar{Y} \bar{Z}')[V(-, \bar{Y}, \bar{Z}') \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \wedge \bar{Z}' =_{\bar{Z}_g} \bar{z}'_g]\} \subseteq \{\bar{z}'_g \mid S_g(\bar{b}_g, \bar{z}_g) \in D\}$$

Thus, we have shown not only that $\{\bar{z}'_g \mid S_g(\bar{b}_g, \bar{z}_g) \in D\}$ does not depend on D, but also that we can rewrite $Q(D \cup r(\bar{a}, \bar{b}))$ as:

$$\begin{aligned} & Q(D \cup r(\bar{a}, \bar{b})) \\ & = V \cup \{(\bar{a}', \bar{b}, \bar{z}') \mid \textstyle\bigwedge_{g \in PART(S(\bar{Y},\bar{Z}),\bar{Z})} (\exists \bar{Y}, \bar{Z}') V(-, \bar{Y}, \bar{Z}') \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \wedge \bar{Z}' =_{\bar{Z}_g} \bar{z}'_g\} \end{aligned}$$

$v(\bar{X}',\bar{Y},\bar{Z}')$		$r(\bar{X},\bar{Y})$	$S_h(\bar{Y}_h,\bar{Z}_h)$	$S_g(\bar{Y}_g,\bar{Z}_g)$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$\bar{x}',\bar{y},\bar{t}'$	$\bar{x}'*,\bar{y}$	$\bar{y}_h,\bar{t}'_h*$	$\bar{b}_g,\bar{t}'_g*$
V	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$\bar{x}',\bar{y},\bar{z}'_g \mid \bar{t}'_h$	$\vdots$	$\vdots$	$\bar{b}_g,\bar{z}'_g*$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3.3: View exposing $\bar{z}'_g$ for any $S_g(\bar{b}_g,\bar{z}'_g*)$ in D.

which tells us precisely how to compute the set of tuples to be inserted into the view in order to keep it up to date.

ONLY-IF:

Assume condition (3.2) is false. We need to construct two database instances D_1 and D_2 that are both consistent with V but such that $Q(D_1 \cup r(\bar{a},\bar{b})) \neq Q(D_2 \cup r(\bar{a},\bar{b}))$.

For D_1, we use the canonical database instance, which we know is consistent with V.

To construct D_2, we add to D_1 a set Δ of new tuples (i.e., tuples not already in D_1) as follows. Since condition (3.2) is not satisfied, some α_g's are false. Then new tuples are included in Δ only for those groups g such that α_g is false. For each such group g, the following specifies how new tuples are added:

Type A If the group has no private variable (i.e. $\bar{Z}_g = \emptyset$), it consists of a single subgoal, say $S_g(\bar{Y}_g)$. It is not difficult to see that by construction of the canonical instance, D_1 could not possibly contain $S_g(\bar{b}_g)$. Therefore we include $S_g(\bar{b}_g)$ in Δ.

Type B If the group has some private variable (i.e. $\bar{Z}_g \neq \emptyset$), Consider $S_g(\bar{b}_g,\bar{z}_g^{new})$ where $\bar{z}_g^{new}$[2] is a vector of new constants with the same arity as $\bar{Z}_g$. Since every atom in $S_g(\bar{b}_g,\bar{z}_g^{new})$ contains a new constant, it cannot be in D_1. We therefore include $S_g(\bar{b}_g,\bar{z}_g^{new})$ in Δ.

This construction of Δ is illustrated in Table 3.4.

Now that we have specified D_2, we need to verify that it is indeed consistent with V. Since $D_1 \subseteq D_2$ and Q is monotonic, we only need to make sure that Q cannot generate any new tuple when Δ is added to D_1. Any new tuple Q generates must use some tuple $t \in \Delta$ which falls into one of the two cases:

- Group g is of type A: t uses $\bar{b}_g$ for its $\bar{Y}_g$ components, but r has no tuples that agree with $\bar{b}$ over $\bar{Y}_g$ (recall this is a consequence of α_g false). Thus, using t, Q cannot generate any V-tuple.

[2]In the remainder of this chapter, we will use the same convention that $\bar{z}_g^{new}$ denotes a vector of constants that appear nowhere else.

	g of type A α_g false	g of type B α_g false	g where α_g true
D_1	$S_g(\bar{b}_g)$ is absent.		Some $S_g(\bar{b}_g, \bar{z}_g)$ present
Δ	Add $S_g(\bar{b}_g)$	Add $S_g(\bar{b}_g, \bar{z}_g^{new})$	No tuples added.

Table 3.4: Counterexample database $D_2 = D_1 \cup \Delta$ in the proof of Theorem 3.2.1.

- Group g is of type B: using some tuple t from $S_g(\bar{b}_g, \bar{z}_g^{new})$ forces us to use all tuples from $S_g(\bar{b}_g, \bar{z}_g^{new})$. But r has no tuples that agree with $\bar{b}$ over $\bar{Y}_g$. So again, using t, Q cannot generate any V-tuple.

Finally, to verify that $Q(D_1 \cup r(\bar{a}, \bar{b})) \neq Q(D_2 \cup r(\bar{a}, \bar{b}))$, we need to find a tuple in $Q(D_2 \cup r(\bar{a}, \bar{b}))$ that is not in $Q(D_1 \cup r(\bar{a}, \bar{b}))$. Consider the tuple t' that results from joining the following tuples from $Q(D_2 \cup r(\bar{a}, \bar{b}))$:

- $r(\bar{a}, \bar{b})$,
- All tuples from Δ,
- For each group g such that α_g holds, there is some value $\bar{z}_g$ that satisfies $S_g(\bar{b}_g, \bar{z}_g)$. We use all the tuples in $S_g(\bar{b}_g, \bar{z}_g)$. Note that these tuples are in both D_1 and D_2.

Let g be a group such that α_g is false.

- If some private variables in g are exposed, then t' cannot be in $Q(D_1 \cup r(\bar{a}, \bar{b}))$ since the $\bar{Z}_g$ components of t' are new constants that do not occur anywhere in D_1.
- If all private variables in g are hidden, then D_1 cannot possibly contain $S_g(\bar{b}_g, -)$ since α_g is false. And since t' agrees with $\bar{b}$ over the $\bar{Y}_g$ components, t' cannot be in $Q(D_1 \cup r(\bar{a}, \bar{b}))$.

■

EXAMPLE 3.2.3 Consider the view definition

$$v(U, Y, W, Z) \text{ :- } r(X, U, Y, W) \ \&\ p_1(U, Y) \ \&\ p_2(Y, Z) \ \&\ p_3(W, Z) \ \&\ p_4(T)$$

and consider the insertion of $r(x, a, b, c)$. In this view definition, the join variables U,Y,W all appear in the head. The subgoals with predicates p_1, p_2, p_3, and p_4 are partitioned into

$v(U,Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
a,b,c',z' a',b,c,z'' a'',b,c,z'''	$x_1,a',b,c^{\ddagger}$	$a,b^{\dagger}$ $a',b^{\ddagger\ddagger}$			$t_1^{\dagger\dagger}$

Figure 3.5: A self-maintainable view instance from Example 3.2.3.

.

three groups: $\{p_1(U,Y)\}$, $\{p_2(Y,Z), p_3(W,Z)\}$, and $\{p_4(T)\}$. Applying Theorem 3.2.1, a given instance V is self-maintainable under the insertion if and only if:

$$V(a,b,-,-) \wedge V(-,b,c,-) \wedge V(- - - -)$$

Note that the last conjunct, which says V must be nonempty, can be dropped since it is implied by the other two conjuncts. To maintain an instance V that is self-maintainable, insert the following tuples:

$$\{(a,b,c,z) \mid V(-,b,c,z)\}$$

Let us illustrate how things work with two instances of V.

First, consider an instance of V that has tuples of both forms $(a,b,-,-)$ and $(-,b,c,-)$, as shown in Figure 3.5.

In Figure 3.5, the first column represents the tuples in the instance of V. The remaining columns represent what can be inferred about the base relations R, P_1, P_2, P_3, and P_4. Note that the subscripted symbols, such as x_1 and t_1, represent constants that exist but whose value is not known exactly. We only show the minimum amount of information inferred that is sufficient to determine the required insertions to V. To see how we can answer the following query unambiguously:

$$\text{Find all } Z\text{: } p_1(a,b) \ \wedge \ p_2(b,Z) \ \wedge \ p_3(c,Z) \ \wedge \ p_4(-)$$

we use † and †† from Figure 3.5 to satisfy the first and last conjuncts. Furthermore, any tuples that satisfy the second and third conjuncts would have joined with ‡, ‡‡ and †† to derive the V-tuple (a',b,c,Z). Conversely, the presence of any V-tuple $(-,b,c,Z)$ implies the presence of tuples in relations P_2 and P_3 that satisfy the second and third conjuncts. Thus, the following query can be used to exactly determine the required insertions to V:

$$\text{Find all } Z\text{: } v(-,b,c,Z)$$

In other words, the view instance given in Figure 3.5 is self-maintainable under the insertion of $r(x,a,b,c)$ and to maintain V, insert tuples (a,b,c,z'') and (a,b,c,z''').

We now consider another instance of V that has tuples of the form $(a,b,-,-)$ but not $(-,b,c,-)$, as shown in Figure 3.6, where the first table shows a database instance D_1 and the view, before and after the insertion, and the second table shows another database instance D_2 and the view, before and after the insertion.

$v(U,Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
a,b,c',z	x_1,a,b,c'	a,b	b,z	c',z	t_1
Nothing added.	Add x,a,b,c				

Canonical database D_1.

$v(U,Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
a,b,c',z	x_1,a,b,c'	a,b	b,z b,z_1	c',z c,z_1	t_1
a,b,c,z_1 added.	Add x,a,b,c				

Database instance D_2 derives differently from D_1 after the insertion.

Figure 3.6: A non-self-maintainable view instance from Example 3.2.3.

.

The base instances D_1 and D_2 are a counterexample showing that the view instance is not self-maintainable under the insertion of $r(x,a,b,c)$. Even though there may be other counterexamples, we choose this one to follow the general construction method presented in the proof. In particular, D_1 is the canonical database, and D_2 is obtained by adding to D_1 the tuples $p_2(b,z_1)$ and $p_3(c,z_1)$, which correspond to the group with the missing $v(-,b,c,-)$, as Table 3.4 shows. $\square$

3.2.3 No Group has both Hidden Join and Exposed Private Variables

Referring to the generic view definition (3.1), this subsection deals with one of the two cases where $\bar{Y}' \subset \bar{Y}$. To emphasize the fact that $\bar{Y}'$ is a strict subset of $\bar{Y}$ in this subsection, we will use $\bar{Y}''$ instead of $\bar{Y}'$.

Theorem 3.2.2 *Let view V be defined by $v(\bar{X}',\bar{Y}'',\bar{Z}')$:- $r(\bar{X},\bar{Y}),S(\bar{Y},\bar{Z})$, where $\bar{X}' \subseteq \bar{X}$, $\bar{Y}'' \subset \bar{Y}$, and $\bar{Z}' \subseteq \bar{Z}$, and where no group in PART$(S(\bar{Y},\bar{Z}),\bar{Z})$ has both hidden join variables and exposed private variables. Then V is self-maintainable under the insertion of $r(\bar{a},\bar{b})$ if and only if V has some tuple $(\bar{a}',\bar{b}'',-)$. Furthermore, in this situation, V is not affected by the insertion of $r(\bar{a},\bar{b})$.* $\square$

Proof:

When some join variables are hidden, it is useful to classify each group g in $PART(S(\bar{Y},\bar{Z}),\bar{Z})$ according to whether or not

- It has private variables: $\bar{Z}_g = \emptyset$.
- All its private variables are hidden: $\bar{Z}'_g = \emptyset$ (where $\bar{Z}'_g$ denotes $\bar{Z}_g \cap \bar{Z}'$).

Type of group g	Syntactic characterization
A_1	$\bar{Z}_g = \emptyset$,$\bar{Y}_g \neq \emptyset$,$\bar{Y}_g \not\subseteq \bar{Y}''$
A_2	$\bar{Z}_g = \emptyset$,$\bar{Y}_g \neq \emptyset$,$\bar{Y}_g \subseteq \bar{Y}''$
AB	$\bar{Y}_g = \emptyset$
B_1	$\bar{Z}_g \neq \emptyset$,$\bar{Y}_g \neq \emptyset$, $\bar{Z}'_g \neq \emptyset$, $\bar{Y}_g \not\subseteq \bar{Y}''$
B_2	$\bar{Z}_g \neq \emptyset$,$\bar{Y}_g \neq \emptyset$, $\bar{Z}'_g \neq \emptyset$, $\bar{Y}_g \subseteq \bar{Y}''$
B_3	$\bar{Z}_g \neq \emptyset$,$\bar{Y}_g \neq \emptyset$, $\bar{Z}'_g = \emptyset$, $\bar{Y}_g \not\subseteq \bar{Y}''$
B_4	$\bar{Z}_g \neq \emptyset$,$\bar{Y}_g \neq \emptyset$ $\bar{Z}'_g = \emptyset$, $\bar{Y}_g \subseteq \bar{Y}''$

Table 3.5: Group types for a view definition with hidden join variables.

- It has join variables: $\bar{Y}_g = \emptyset$.
- All its join variables are exposed: $\bar{Y}_g \subseteq \bar{Y}''$.

Table 3.5 shows the classification of groups into seven types: A_1, A_2, AB, B_1, B_2, B_3, or B_4. This classification will be used later in the proofs.

In this proof, since no group has both hidden join variables (i.e., $\bar{Y}_g \not\subseteq \bar{Y}''$) and exposed private variables (i.e., $\bar{Z}'_g \neq \emptyset$), we will not be using type B_1.

IF: Assume V has some $(\bar{a}', \bar{b}'', -)$ tuple. Let D be an arbitrary database instance consistent with V. We need to show that $Q(D \cup r(\bar{a}, \bar{b}))$ does not depend on D.

$$\begin{aligned}
&Q(D \cup r(\bar{a}, \bar{b}))\\
&= Q(D) \cup \{(\bar{a}', \bar{b}'', \bar{z}') \mid S(\bar{b}, \bar{z}) \in D\}\\
&= V \cup \{(\bar{a}', \bar{b}'')\} \times \{(\bar{z}') \mid \textstyle\bigwedge_g S_g(\bar{b}_g, \bar{z}_g) \in D\}
\end{aligned}$$

We observe that only groups of type B_2 or AB can contribute any value to the exposed $\bar{Z}$-variables, since $\bar{Z}'_g = \emptyset$ for all other groups g. To reflect this observation, let G denote all the groups of type B_2 or AB, and H the remaining groups. $S_G(\bar{Y}_G, \bar{Z}_G)$ denotes the set of subgoals in G, and $S_H(\bar{Y}_H, \bar{Z}_H)$ the remaining subgoals. Note that $\bar{Z}' \subseteq \bar{Z}_G$, $\bar{Y}_G \subseteq \bar{Y}''$, and $\bar{Z}_H \cap \bar{Z}' = \emptyset$. We then rewrite $Q(D \cup r(\bar{a}, \bar{b}))$ as:

$$V \cup \{(\bar{a}', \bar{b}'')\} \times \{(\bar{z}') \mid S_G(\bar{b}_G, \bar{z}_G) \in D\} \times \{() \mid S_H(\bar{b}_H, \bar{z}_H) \in D\}$$

The last factor in the cross-product represents a condition (boolean query) that can be either true or false, depending on the actual instance of D. However, it does not matter whether the condition is true or false, since we will show that the following is contained in V, and hence that $Q(D \cup r(\bar{a}, \bar{b})) = V$, regardless of D:

$$\{(\bar{a}', \bar{b}'')\} \times \{(\bar{z}') \mid S_G(\bar{b}_G, \bar{z}_G) \in D\} \tag{3.3}$$

In fact, since V has some $(\bar{a}', \bar{b}'', -)$ tuple by hypothesis, this tuple must be derived from some set of tuples in D, each of which agrees with $\bar{b}''$. Call this set T. We claim that any

$v(\bar{X}',\bar{Y}'',\bar{Z}')$		$r(\bar{X},\bar{Y})$	$S_H(\bar{Y}_H,\bar{Z}_H)$	$S_G(\bar{Y}_G,\bar{Z}_G)$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$\bar{a}',\bar{b}'',\bar{t}'$	$\bar{a}'*,\bar{b}''+$	$\bar{b}''_H+,\bar{t}'_H*$	$\bar{b}''_G,\bar{t}'_G*$
V	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	$\bar{a}',\bar{b}'',\bar{z}'_G\vert\bar{t}'_H$	$\vdots$	$\vdots$	$\bar{b}''_G,\bar{z}'_G*$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Table 3.6: View exposing $\bar{z}'_G$ for any $S_G(\bar{b}_G,\bar{z}'_G*)$ in D.

set of tuples $S_G(\bar{b}_G,\bar{z}_G)$ in D would join with T to derive a V-tuple $(\bar{a}',\bar{b}'',\bar{z}')$, where $\bar{z}'$ agrees with all the $\bar{z}_g$'s. This situation is depicted in Table 3.6.

In other words, the following containment holds:

$$\{(\bar{z}') \mid S_G(\bar{b}_G,\bar{z}_G)\in D\} \subseteq \{(\bar{z}') \mid V(\bar{a}',\bar{b}'',\bar{z}')\}$$

and consequently, (3.3) is contained in V.

We conclude that $Q(D\cup r(\bar{a},\bar{b})) = V$, and $Q(D\cup r(\bar{a},\bar{b}))$ is independent of D. View V is self-maintainable under the insertion of $r(\bar{a},\bar{b})$ simply because *no change is needed* to bring it up to date.

ONLY-IF: Assume V has no tuples of the form $(\bar{a}',\bar{b}'',-)$. We need to construct two database instances D_1 and D_2 that are both consistent with V prior to inserting $r(\bar{a},\bar{b})$ but that derive differently after the insertion.

We can assume $V\neq\emptyset$, since otherwise, a trivial counterexample can be constructed.

Let D_1 be the canonical database. D_1 is consistent with V. Furthermore, since $\bar{Y}''\subset\bar{Y}$, D_1 has no $S(\bar{b},-)$. As a consequence, the newly inserted $r(\bar{a},\bar{b})$ cannot join with S, and therefore $Q(D_1\cup r(\bar{a},\bar{b})) = Q(D_1)$.

To construct D_2, a set Δ of new tuples is added to D_1. Δ is specified in Table 3.7. Note that type B_1 is omitted since there are no groups of this type, as mentioned at the beginning of the proof.

To show that D_2 is consistent with V, we show that no tuple in $Q(D_2)$ can be derived using some tuple $t\in\Delta$. The following considers all possible cases tuple t can be in:

- Type A_1. D_1 has no $r(-,-\bar{b}_g-)$ since $\bar{Y}_g$ is not completely exposed. Therefore t cannot join with any tuple from r.

- Type A_2. D_1 has no $r(-,-\bar{b}_g-)$ since V has no $(-,-\bar{b}_g-,-)$. Therefore t cannot join with any tuple from r.

- Type B_2. D_1 has no $r(-,-\bar{b}_g-)$ since V has no $(-,-\bar{b}_g-,-)$. Using any tuple t from $S_g(\bar{b}_g,\bar{z}_g^{new})$ forces all tuples from $S_g(\bar{b}_g,\bar{z}_g^{new})$ to be used. So S_g generates exactly the tuple $(\bar{b}_g,\bar{z}_g^{new})$, which cannot join with any tuple from r.

	Type A_1	Type A_2	Type AB
D_1	$S_g(\bar{b}_g)$ absent		Some $S_g(-)$ already present
Δ	Add $S_g(\bar{b}_g)$	Add $S_g(\bar{b}_g)$ if no $V(-,-\bar{b}_g-,-)$	No tuples added

	Type B_2	Type B_3	Type B_4
D_1			
Δ	Add $S_g(\bar{b}_g, \bar{z}_g^{new})$ if no $V(-,-\bar{b}_g-,-)$	Add $S_g(\bar{b}_g, \bar{z}_g^{new})$	Add $S_g(\bar{b}_g, \bar{z}_g^{new})$ if no $V(-,-\bar{b}_g-,-)$

Table 3.7: Counterexample database $D_2 = D_1 \cup \Delta$ in the proof of Theorem 3.2.2.

- Type B_3. D_1 has no $r(-,-\bar{b}_g-)$ since $\bar{Y}_g$ is not completely exposed. Using any tuple t from $S_g(\bar{b}_g, \bar{z}_g^{new})$ forces all tuples from $S_g(\bar{b}_g, \bar{z}_g^{new})$ to be used. So S_g generates exactly the tuple $(\bar{b}_g, \bar{z}_g^{new})$, which cannot join with any tuple from r.
- Type B_4: same arguments as for type B_2.

Finally, to show that D_1 and D_2 derive different views after the insertion of $r(\bar{a}, \bar{b})$, we will find a tuple $t' \in Q(D_2 \cup r(\bar{a}, \bar{b}))$ that is not in V. Consider the tuple $t' = (\bar{a}', \bar{b}'', \bar{z}')$ derived by joining the following tuples from $Q(D_2 \cup r(\bar{a}, \bar{b}))$:

- $r(\bar{a}, \bar{b})$,
- All tuples from Δ,
- For each group g that contributes no tuples to Δ, there is some value $\bar{z}_g$ that satisfies $S_g(\bar{b}_g, \bar{z}_g)$. We use all the tuples in $S_g(\bar{a}_g, \bar{z}_g)$. Note that these tuples are in both D_1 and D_2.

Now, t' cannot possibly be in V since by hypothesis, V has no tuples of the form $(\bar{a}', \bar{b}'', -)$. ∎

EXAMPLE 3.2.4 Consider the view definition

$$v(Y,W,Z) :- r(X,U,Y,W) \ \& \ p_1(U,Y) \ \& \ p_2(Y,Z) \ \& \ p_3(W,Z) \ \& \ p_4(T)$$

and consider the insertion of $r(x,a,b,c)$. Note this view definition is almost similar to that in Example 3.2.3, with the only exception that the join variable U is projected out of the head. The nonupdated subgoals are partitioned into three groups: $\{p_1(U,Y)\}$, $\{p_2(Y,Z), p_3(W,Z)\}$, and $\{p_4(T)\}$. The first group has a hidden join variable (namely U), but has no private variables. In the second group, all the join variables are exposed (namely Y and W). The third group has no join variables. Applying Theorem 3.2.2, a given instance V is self-maintainable under the insertion if and only if:

$$V(b,c,-)$$

$v(Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
b,c',z' b,c,z'' b,c,z'''	$x_1,a_1,b,c^{\dagger}$	$a_1,b^{\dagger\dagger}$			$t_1^{\dagger\dagger\dagger}$

Figure 3.7: A self-maintainable view instance from Example 3.2.4.

.

and in this situation, V is not affected by the insertion.

First, consider an instance of V that has tuples of the form $(b,c,-)$, as shown in Figure 3.7. To determine the required insertions to V, we consider the following query:

$$\text{Find all } Z\text{: } p_1(a,b) \;\wedge\; p_2(b,Z) \;\wedge\; p_3(c,Z) \;\wedge\; p_4(-)$$

Like the self-maintainable case in Example 3.2.3, the last conjunct is satisfied by $\dagger\dagger\dagger$ (that is, the presence of tuple $p_4(t_1)$) from Figure 3.7. And also like Example 3.2.3, all the values of Z that satisfy the second and third conjuncts can be found in the view instance, namely z'' andz'''. But unlike Example 3.2.3, we cannot determine whether the first conjunct is satisfied or not. Relation P_1 might or might not contain (a,b), and both cases are consistent with the given view instance. Fortunately, this does not matter because in either case, we are not inserting anything new into V: if (a,b) is not in P_1, we are not inserting anything into V; if (a,b) is in P_1, we insert into V at most (b,c,z'') and (b,c,z'''); but these tuples are already in V. Therefore, the view instance shown in Figure 3.7 is self-maintainable under the insertion of $r(x,a,b,c)$ simply because the view is not affected by the insertion.

We now consider another instance of V that does not have tuples of the form $(b,c,-)$, as shown in Figure 3.8.

This counterexample mirrors the general construction method presented in the proof. In particular, the three groups $\{p_1(U,Y)\}$, $\{p_2(Y,Z), p_3(W,Z)\}$, and $\{p_4(T)\}$ are of type A_1, B_2, and AB respectively. According to Table 3.7, D_2 is obtained from D_1 by adding only tuples for groups of type A_1 (namely $r(a,b)$), and tuples for groups of type B_2 (namely $p_2(b,z_1)$ and $p_3(c,z_1)$). □

3.2.4 Some Group has both Hidden Join and Exposed Private Variables

Referring to the generic view definition (3.1), this subsection deals with one of the two cases where $\bar{Y}' \subset \bar{Y}$. To emphasize the fact that $\bar{Y}'$ is a strict subset of $\bar{Y}$ in this subsection, we will use $\bar{Y}''$ instead of $\bar{Y}'$.

Theorem 3.2.3 *Let view V be defined by $v(\bar{X}',\bar{Y}'',\bar{Z}')$:- $r(\bar{X},\bar{Y}), S(\bar{Y},\bar{Z})$, where $\bar{X}' \subseteq \bar{X}$, $\bar{Y}'' \subset \bar{Y}$, and $\bar{Z}' \subseteq \bar{Z}$, and where some group in $PART(S(\bar{Y},\bar{Z}),\bar{Z})$ has both hidden join variables and exposed private variables. Then V is not self-maintainable under the insertion of $r(\bar{a},\bar{b})$.* □

$v(Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
b,c',z	x_1,a_1,b,c'	a_1,b	b,z	c',z	t_1
Nothing added.	Add x,a,b,c				

Canonical database D_1.

$v(Y,W,Z)$	$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
b,c',z	x_1,a_1,b,c'	a_1,b a,b	b,z b,z_1	c',z c,z_1	t_1
b,c,z_1 added.	Add x,a,b,c				

Database instance D_2 derives differently from D_1 after the insertion.

Figure 3.8: A non-self-maintainable view instance from Example 3.2.4.

.

Proof:

We need to show we can always find two database instances D_1 and D_2 that are both consistent with V before the insertion of $r(\bar{a},\bar{b})$ but that derive differently after the insertion.

We can assume $V \neq \emptyset$, since otherwise a trivial counterexample can be constructed.

Let D_1 be the canonical database. D_1 is consistent with V. Since $\bar{Y}'' \subset \bar{Y}$, D_1 has no $r(-,\bar{b})$ and no $S(\bar{b},-)$. As a consequence, the newly inserted $r(\bar{a},\bar{b})$ cannot join with S, and therefore $Q(D_1 \cup r(\bar{a},\bar{b})) = Q(D_1)$.

To construct D_2, a set Δ of new tuples is added to D_1. Δ is specified in Table 3.8.

To show that D_2 is consistent with V, we show that no tuple in $Q(D_2)$ can be derived using some tuple $t \in \Delta$. The following considers all possible cases tuple t can be in:

	Type A_1	Type A_2	Type AB
D_1	$S_g(b_g)$ absent		Some $S_g(-)$ already present
Δ	Add $S_g(\bar{b}_g)$	Add $S_g(\bar{b}_g)$ if no $V(-,-\bar{b}_g-,-)$	No tuples added

	Type B_1	Type B_2	Type B_3	Type B_4
D_1				
Δ	Add $S_g(\bar{b}_g,\bar{z}_g^{new})$	Add $S_g(\bar{b}_g,\bar{z}_g^{new})$ if no $V(-,-\bar{b}_g-,-)$	Add $S_g(\bar{b}_g,\bar{z}_g^{new})$	Add $S_g(\bar{b}_g,\bar{z}_g^{new})$ if no $V(-,-\bar{b}_g-,-)$

Table 3.8: Counterexample database $D_2 = D_1 \cup \Delta$ in the proof of Theorem 3.2.3.

- Type A_1. D_1 has no $r(-,-\bar{b}_g-)$ since $\bar{Y}_g$ is not completely exposed. Therefore t cannot join with any tuple from r.
- Type A_2. D_1 has no $r(-,-\bar{b}_g-)$ since V has no $(-,-\bar{b}_g-,-)$. Therefore t cannot join with any tuple from r.
- Type B_1. D_1 has no $r(-,-\bar{b}_g-)$ since $\bar{Y}_g$ is not completely exposed. Using any tuple t from $S_g(\bar{b}_g, \bar{z}_g^{new})$ forces all tuples from $S_g(\bar{b}_g, \bar{z}_g^{new})$ to be used. So S_g generates exactly the tuple $(\bar{b}_g, \bar{z}_g^{new})$ which cannot join with any tuple from r.
- Type B_2. D_1 has no $r(-,-\bar{b}_g-)$ since V has no $(-,-\bar{b}_g-,-)$. Using any tuple t from $S_g(\bar{b}_g, \bar{z}_g^{new})$ forces all tuples from $S_g(\bar{b}_g, \bar{z}_g^{new})$ to be used. So S_g generates exactly the tuple $(\bar{b}_g, \bar{z}_g^{new})$ which cannot join with any tuple from r.
- Type B_3: same arguments as for type B_1.
- Type B_4: same arguments as for type B_2.

Finally, to show that D_1 and D_2 derive different views after the insertion of $r(\bar{a}, \bar{b})$, we will find a tuple $t' \in Q(D_2 \cup r(\bar{a}, \bar{b}))$ that is not in V. Consider the tuple $t' = (\bar{a}', \bar{b}'', \bar{z}')$ derived by joining the following tuples from $Q(D_2 \cup r(\bar{a}, \bar{b}))$:

- $r(\bar{a}, \bar{b})$,
- All the new facts from Δ (there is a least one such fact, since there is at least a group of type B_1),
- For each group g that contributes no tuples to Δ, there is some value $\bar{z}_g$ that satisfies $S_g(\bar{b}_g, \bar{z}_g)$. We use all the tuples in $S_g(\bar{b}_g, \bar{z}_g)$. Note that these tuples are in both D_1 and D_2.

Now, t' cannot possibly be in V since it is derived from some tuples added under type B_1, and hence must have components with new values (recall that a group of type B_1 has some $\bar{Z}$-variables exposed). ∎

EXAMPLE 3.2.5 Consider the view definition

$$v(W, Z) :- r(X, U, Y, W) \ \& \ p_1(U, Y) \ \& \ p_2(Y, Z) \ \& \ p_3(W, Z) \ \& \ p_4(T)$$

and consider the insertion of $r(x, a, b, c)$. Note this view definition is almost similar to that in Example 3.2.4, with the only exception that an additional join variable, Y, is projected out of the head. The nonupdated subgoals are partitioned into three groups: $\{p_1(U, Y)\}$, $\{p_2(Y, Z), p_3(W, Z)\}$, and $\{p_4(T)\}$. The second group has a hidden join variable (namely Y) but also an exposed private variable (namely Z). Applying Theorem 3.2.3, no instance of V is self-maintainable under the insertion.

In the following, we explain why updating view V is inherently ambiguous, no matter what its contents are.

$v(W,Z)$			$r(X,U,Y,W)$	$p_1(U,Y)$	$p_2(Y,Z)$	$p_3(W,Z)$	$p_4(T)$
	$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
V	w,z	D_1	x',u',y',w	u',y'	y',z	w,z	t'
	$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
		Δ		a,b	b,z'	c,z'	

Figure 3.9: No instance of view as defined in Example 3.2.5 is self-maintainable under the insertion of $r(x,a,b,c)$.

Figure 3.9 shows a counterexample for an arbitrary view instance. In the figure, x', u', y', and t' represent new symbols that are created for each line, and z' is another new symbol. Since no R-tuple can have b in its Y component, Δ cannot contribute to the view, and D_2 is consistent with V. Furthermore, adding $r(x,a,b,c)$ to D_1 does not affect the view since D_1 has no $p_1(a,b)$. But adding $r(x,a,b,c)$ to D_2 contributes at least the tuple (c,z') to V. Thus, we can always find a counterexample, no matter what the contents of V are and no matter what tuple is inserted into R.

Note that the counterexample is based on the general construction method presented in the proof. In particular, the three groups $\{p_1(U,Y)\}$, $\{p_2(Y,Z), p_3(W,Z)\}$, and $\{p_4(T)\}$ are of type A_1, B_1, and AB respectively. According to Table 3.9, D_2 is obtained from D_1 by only adding tuples for groups of type A_1 (namely $p_1(a,b)$), and tuples for groups of type B_1 (namely $p_2(b,z')$ and $p_3(c,z')$). $\square$

3.2.5 Multiple Insertions

Theorem 3.2.4 *Let view V be defined by $v(\bar{X}',\bar{Y}',\bar{Z}')$:- $r(\bar{X},\bar{Y})$ & $S(\bar{Y},\bar{Z})$, where $\bar{X}' \subseteq \bar{X}$, $\bar{Y}' \subseteq \bar{Y}$, and $\bar{Z}' \subseteq \bar{Z}$. Let U be a base update that consists of the insertion of tuples $r(\bar{x}_1,\bar{y}_1),\ldots,r(\bar{x}_n,\bar{y}_n)$. Then V is self-maintainable under U if and only if for every $i = 1,\ldots,n$, V is self-maintainable under the insertion of $r(\bar{x}_i,\bar{y}_i)$. Furthermore, if V is self-maintainable under U, to maintain V under U, maintain V under the insertion of $r(\bar{x}_i,\bar{y}_i)$ for each i.* $\square$

Proof:

Let Q denote the conjunctive query defining view V. Recall that we assume no predicate is repeated in Q.

IF: If V is self-maintainable under the insertion of $r(\bar{x}_i,\bar{y}_i)$, then $Q(D \cup r(\bar{x}_i,\bar{y}_i))$ does not depend on D. Since Q has only one subgoal with predicate r (that is, r does not occur in S), it follows that $Q(D \cup U) = \bigcup_i Q(D \cup r(\bar{x}_i,\bar{y}_i))$. Therefore, $Q(D \cup U)$ does not depend on D.

ONLY-IF: We need to consider two cases: the case where all join variables are exposed, and the case where some join variables are hidden.

In the first case, the view is defined by $v(\bar{X}', \bar{Y}, \bar{Z}')$:- $r(\bar{X}, \bar{Y})$ & $S(\bar{Y}, \bar{Z})$. Suppose V is not self-maintainable under the insertion of $r(\bar{x}_k, \bar{y}_k)$, for some k. Then there must be two instances D_1 and D_2, both consistent with V, and some value $\bar{z}'$ such that $(\bar{x}'_k, \bar{y}_k, \bar{z}')$ is in $Q(D_1 \cup r(\bar{x}_k, \bar{y}_k))$ but not in $Q(D_2 \cup r(\bar{x}_k, \bar{y}_k))$. There is no $\bar{z}$ that extends $\bar{z}'$ to the remaining components in $\bar{Z}$ such that $S(\bar{y}_k, \bar{z})$ is in $Q(D_2 \cup r(\bar{x}_k, \bar{y}_k))$. This fact remains true in $D_2 \cup U$, since U only affects relation R. Therefore $(\bar{x}'_k, \bar{y}_k, \bar{z}')$ cannot be in $Q(D_2 \cup U)$, while it is still in $Q(D_1 \cup U)$. So $Q(D_1 \cup U) \neq Q(D_2 \cup U)$.

In the second case, we prove non-self-maintainability by constructing a counterexample in a manner analogous to the case of single insertions. ∎

3.3 Mixing Insertions with Deletions

In this section, we consider updates that consist of both insertions and deletions to the same base relations. We define an update as a set of tuples to be inserted and another set of tuples to be deleted. For an update thusly defined to be meaningful, we assume that no tuples get both inserted and deleted. No generality is lost in practice because we can always convert a sequence of insertions and deletions with the same tuple repeated many times to a set of insertions and a set of deletions that are disjoint.

So, consider an update $U = \delta^+ r \cup \delta^- r$, and assume $\delta^+ r$ and $\delta^- r$ are disjoint. Then for any database D, $U(D)$ is uniquely defined and can be written either as $(D \cup \delta^+ r) - \delta^- r$ or as $(D - \delta^- r) \cup \delta^+ r$.

3.3.1 All Updated Variables Exposed

Theorem 3.3.1 *Let view V be defined by $v(\bar{X}, \bar{Y}, \bar{Z}')$:- $r(\bar{X}, \bar{Y})$ & $S(\bar{Y}, \bar{Z})$, where $\bar{Z}' \subseteq \bar{Z}$. Let U be an update that consists of $\delta^+ r \cup \delta^- r$. Then V is self-maintainable under U if and only if it is self-maintainable under $\delta^+ r$, To maintain V, insert tuples according to maintenance under the insertion of each tuple in $\delta^+ r$, and delete all tuples in V that join with $\delta^- r$. The order of deletions and insertions to V is immaterial.* □

Proof:

IF: Assume V is self-maintainable under $\delta^+ r$. To show V is self-maintainable under U, consider two arbitrary instances D_1 and D_2 that are consistent with V. Since V is self-maintainable under $\delta^+ r$, $Q(D_1 \cup \delta^+ r)$ and $Q(D_2 \cup \delta^+ r)$ are identical. Let V' be this common view. But any view instance is self-maintainable under $\delta^- r$, in particular V'. In other words, $Q((D_1 \cup \delta^+ r) - \delta^- r)$ is identical to $Q((D_2 \cup \delta^+ r) - \delta^- r)$. This situation is depicted in Figure 3.10.

V' is obtained from V by adding tuples according to the maintenance plan for each and every insertion in $\delta^+ r$. Note that every tuple added to V joins $\delta^+ r$. The final view is obtained from V' by deleting all tuples that join with $\delta^- r$. Since we assume $\delta^+ r$ and $\delta^- r$ are disjoint, the set of tuples added to V and the set of tuples deleted from V are also disjoint. Thus, to determine the final view from V, the order of insertions and deletions is not important.

$$\begin{tikzcd}
D_1 \arrow[r, "Q"] \arrow[d, "\delta^+ r"'] & V & D_2 \arrow[l, "Q"'] \arrow[d, "\delta^+ r"] \\
D_1' \arrow[r] \arrow[d, "\delta^- r"'] & V' & D_2' \arrow[l] \arrow[d, "\delta^- r"] \\
D_1'' \arrow[r] & = & D_2'' \arrow[l]
\end{tikzcd}$$

Figure 3.10: Self-maintainability under insertions and deletions when all updated variables are exposed.

ONLY-IF: Let $\delta^+ r = \{r(\bar{x}_1, \bar{y}_1), \ldots, r(\bar{x}_n, \bar{y}_n)\}$. Assume V is not self-maintainable under $\delta^+ r$. Then there are two databases D_1 and D_2, both consistent with V, and some k and $\bar{z}'$ such that $(\bar{x}_k, \bar{y}_k, \bar{z}') \in Q(D_1 \cup \delta^+ r)$ but $\notin Q(D_2 \cup \delta^+ r)$. On the one hand, $(\bar{x}_k, \bar{y}_k, \bar{z}') \in Q((D_1 \cup \delta^+ r) - \delta^- r)$: since the only effect $\delta^- r$ has over $Q(D_1 \cup \delta^+ r)$ is to delete $(\bar{x}, \bar{y}, -)$ such that $(\bar{x}, \bar{y}) \in \delta^- r$, and since $\delta^+ r$ and $\delta^- r$ are disjoint, $(\bar{x}_k, \bar{y}_k, \bar{z}')$ cannot be deleted. On the other hand, $(\bar{x}_k, \bar{y}_k, \bar{z}') \notin Q((D_2 \cup \delta^+ r) - \delta^- r)$, since Q is monotonic. Thus, $Q(U(D_1) \neq Q(U(D_2)$. ∎

3.3.2 Some Updated Variables Hidden

Theorem 3.3.2 *Let view V be defined by $v(\bar{X}', \bar{Y}', \bar{Z}')$:- $r(\bar{X}, \bar{Y})$ & $S(\bar{Y}, \bar{Z})$, where $\bar{X}' \subseteq \bar{X}$, $\bar{Y}' \subseteq \bar{Y}$, and $\bar{Z}' \subseteq \bar{Z}$. Further, assume that either $\bar{X}' \neq \bar{X}$ or $\bar{Y}' \neq \bar{Y}$. Let U be an update that consists of $\delta^+ r \cup \delta^- r$. Then V is self-maintainable under U if it is self-maintainable under each of $\delta^+ r$ and $\delta^- r$.* □

Proof:
Assume V is self-maintainable under each of $\delta^+ r$ and $\delta^- r$. To show V is self-maintainable under U, let D_1 and D_2 be consistent with V. Since V is self-maintainable under $\delta^- r$, it follows from Theorem 3.1.5 that $Q(D_1 - \delta^- r) = Q(D_2 - \delta^- r) = V$. And since V is self-maintainable under $\delta^+ r$, $Q((D_1 - \delta^- r) \cup \delta^+ r) = Q((D_2 - \delta^- r) \cup \delta^+ r)$ follows. This situation is depicted in Figure 3.11.

∎

However, while the converse also holds in certain cases, it does not generally hold, as stated in the following theorem.

Theorem 3.3.3 *Let view V be defined by $v(\bar{X}', \bar{Y}', \bar{Z}')$:- $r(\bar{X}, \bar{Y})$ & $S(\bar{Y}, \bar{Z})$, where $\bar{X}' \subseteq \bar{X}$, $\bar{Y}' \subseteq \bar{Y}$, and $\bar{Z}' \subseteq \bar{Z}$. Furthermore, either $\bar{X}' \neq \bar{X}$ or $\bar{Y}' \neq \bar{Y}$. Let U be an update that consists of $\delta^+ r \cup \delta^- r$. If the projection of $\delta^+ r$ over $\bar{X}'$ and $\bar{Y}'$ and the projection of $\delta^- r$*

$$\begin{tikzcd}
D_1 \arrow[r, "Q"] \arrow[d, "\delta^- r"'] & V & D_2 \arrow[l, "Q"'] \arrow[d, "\delta^- r"] \\
D_1' \arrow[r] \arrow[d, "\delta^+ r"'] & V & D_2' \arrow[l] \arrow[d, "\delta^+ r"] \\
D_1'' \arrow[r] & = & D_2'' \arrow[l]
\end{tikzcd}$$

Figure 3.11: Self-maintainability under insertions and deletions when some updated variables are hidden.

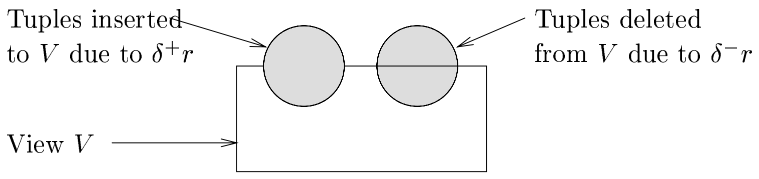

Tuples inserted to V due to δ^+r
Tuples deleted from V due to δ^-r
View V

Figure 3.12: Insertions and deletions have independent effects on the view.

over $\bar{X}'$ and $\bar{Y}'$ do not share any tuple, then V is self-maintainable under U only if it is self-maintainable under each of δ^+r and δ^-r. However, this implication does not generally hold if the projections share some tuples. □

Proof:

Suppose the projection of δ^+r over $\bar{X}'$ and $\bar{Y}'$ and the projection of δ^-r over $\bar{X}'$ and $\bar{Y}'$ do not share any tuple. Then the effect of δ^+r on the view and the effect of δ^-r on the view are independent, since any tuple that can be potentially added to the view must agree with some tuple from δ^+r over $\bar{X}'$ and $\bar{Y}'$, and any tuple that can be potentially deleted from the view must agree with some tuple from δ^-r over $\bar{X}'$ and $\bar{Y}'$. Thus, if V is not self-maintainable under either δ^+r or δ^-r, it cannot be self-maintainable under $\delta^+r \cup \delta^-r$. This situation is depicted in Figure 3.12.

To show that the implication does not generally hold when the projections have some tuples in common, we need to find a view definition, an instance of V, and an instance of δ^+r and δ^-r, such that V is self-maintainable under $\delta^+r \cup \delta^-r$ but not self-maintainable under either δ^+r or δ^-r.

We first consider a view definition that has some hidden join variables:

$$v(Z) \coloneq r(X, Y, Z) \;\&\; s(Y, Z)$$

Consider the view instance $V = \{(c)\}$, the insertion of $r(a', b, c)$, and the deletion of $r(a, b, c)$. V is clearly not self-maintainable under the insertion of $r(a', b, c)$ (in fact no view instance is self-maintainable under insertions to R) or the deletion of $r(a, b, c)$. But we claim that V is self-maintainable under both the insertion and deletion. To see why, consider the following cases:

- If $s(b, c)$ holds, since the final database contains $r(a', b, c)$, the view is guaranteed to remain unchanged.
- If $s(b, c)$ does not hold, the database must contain some tuples $r(x, y, c)$ and $s(y, c)$ where $y \neq b$. Since the presence of these tuples is not affected by the update, the view is guaranteed to remain unchanged.

We how consider another example that shows that even if all the join variables are exposed, the implication still does not hold in general. Consider the view definition:

$$v(Y, Z) :\!-\ r(X, Y, Z)\ \&\ s(Y, Z)$$

Consider the view instance $V = \{(b, c)\}$, the insertion of $r(a', b, c)$, and the deletion of $r(a, b, c)$. V is clearly not self-maintainable under the deletion of $r(a, b, c)$. But it is self-maintainable under both the insertion and deletion:

- On the one hand, we can infer $s(b, c)$ holds.
- On the other hand, the final database contains $r(a', b, c)$. Therefore, the view is guaranteed to remain unchanged.

■

3.4 Summary

- View self-maintenance under single insertions and single deletions has a simple solution: self-maintainability tests and self-maintenance expressions are simple queries whose size is linear in the size of the view definition. Table 3.9 summarizes these results. The runtime of these queries is linear in the size of the view instance. If the view has the appropriate indexes defined, we can even obtain a constant runtime.
- We then studied the problem of view self-maintenance under sets of updates. The interesting question there is whether or not we loose information if we were to treat the individual updates separately. We showed that for sets of deletions from a single relation, sets of insertions into a single relation, and certain sets of insertions and deletions to a single relation, there is no advantage to treating the updates as a set. However, we also showed certain sets of insertions and deletions to a single relation under which a view a self-maintainable, while it is not so under individual insertions and deletions. In such cases, there is definite advantage to treating the updates as

Case	*Self-maintainability test*	*Maintenance expression*
$\bar{X}' = \bar{X}$ and $\bar{Y}' = \bar{Y}$	*TRUE*	Delete all $(\bar{a}, \bar{b}, -)$ from V.
$\bar{X}' \subset \bar{X}$ or $\bar{Y}' \subset \bar{Y}$	No $V(\bar{a}', \bar{b}', -)$.	No update needed.

Under the deletion of $r(\bar{a}, \bar{b})$.

Case	*Self-maintainability test*	*Maintenance expression*
$\bar{Y}' = \bar{Y}$	$\bigwedge_g (\exists \bar{Y})[V(-, \bar{Y}, -) \wedge \bar{Y} =_{\bar{Y}_g} \bar{b}]$.	Insert $(\bar{a}', \bar{b}, \bar{z}')$ where the $\bar{Z}_g$-components of $\bar{z}'$ are obtained as $\{\bar{z}'_g \mid (\exists \bar{Y}, \bar{Z}') V(-, \bar{Y}, \bar{Z}') \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \wedge \bar{Z}' =_{\bar{Z}_g} \bar{z}'_g\}$
$(\forall g) \bar{Y}_g \subseteq \bar{Y}' \vee \bar{Z}_g \cap \bar{Z}' = \emptyset$	$V(\bar{a}', \bar{b}', -)$.	No update needed.
$(\exists g) \bar{Y}_g \not\subseteq \bar{Y}' \wedge \bar{Z}_g \cap \bar{Z}' \neq \emptyset$	*FALSE.*	Not applicable.

Under the insertion of $r(\bar{a}, \bar{b})$.

Table 3.9: Summary of self-maintenance of view $v(\bar{X}', \bar{Y}', \bar{Z}')$:- $r(\bar{X}, \bar{Y})$ & $S(\bar{Y}, \bar{Z})$.

a set. Furthermore, when the set consists of updates across multiple relations, the benefit of treating the updates as a set becomes even more obvious, as the following example shows. Consider a view defined by:

$$v() \text{ :- } r(X) \ \& \ s(X)$$

and consider the insertion of $r(a)$ and $s(a)$. While the view instance $V = \emptyset$ is clearly not self-maintainable under either insertion, we can determine the new state of the view unambiguously when both insertions are considered: $V = \{()\}$. We will show how to deal with arbitrary mixes of insertions and deletions to multiple base relations in Chapter 5.

- In this chapter, we consider views defined by conjunctive queries with no self-joins. This restriction allows us to find efficient solutions to the view self-maintenance problem. We are naturally led to wonder whether efficient solutions exist if we allow self-joins in the view definitions. While we cannot answer this question in general, we have identified a class of self-joins that admit efficient solutions, called exposed self-joins. A self-join is said to be *exposed* if all subgoals in the self-join use only join variables that are exposed.

  For instance, consider the view definition

$$v(X, Y, T) \text{ :- } r(X, Y, T) \ \& \ s(X, Y, X) \ \& \ s(Y, 1, Y) \ \& \ t(X, Z) \ \& \ u(T, Z)$$

and assume r is the updated predicate. Since the subgoals with predicate s only use variables X and Y which appear in both the head and the r subgoal, the view definition has only exposed self-joins.

The following theorem tells us how to self-maintain, under insertions into a single base relation, a view defined by a conjunctive query where the updated predicate is not repeated and all of whose self-joins are exposed.

Theorem 3.4.1 *Consider a view V defined by*

$$v(\bar{X}',\bar{Y}',\bar{Z}') \text{ :- } r(\bar{X},\bar{Y}) \;\&\; M(\bar{U}) \;\&\; S(\bar{V},\bar{Z}).$$

where $\bar{X}$, $\bar{Y}$, and $\bar{Z}$ are disjoint sets of variables, $\bar{X}' \subseteq \bar{X}$, $\bar{Y}' \subseteq \bar{Y}$, $\bar{Z}' \subseteq \bar{Z}$, $\bar{U} \cup \bar{V} = \bar{Y}$, $\bar{U} \subseteq \bar{Y}'$, M is a conjunction of subgoals whose predicate is repeated, and S is a conjunction of subgoals with unique predicates. Consider the insertion of $r(\bar{a},\bar{b})$. Let us define $\tau_{\bar{b}}$ to be the boolean query

$$\begin{aligned} \tau_{\bar{b}} &\text{ :- } M(\bar{U}) \;\&\; \bar{Y} = \bar{b} \\ M(\bar{U}) &\text{ :- } v(\bar{X}',\bar{Y}',\bar{Z}') \end{aligned}$$

(1) If $\bar{Y} = \bar{Y}'$, V is self-maintainable under the insertion if and only if:

$$\tau_{\bar{b}} \wedge \bigwedge_{g \in PART(S(\bar{V},\bar{Z}),\bar{Z})} (\exists \bar{Y}) V(-,\bar{Y},-) \wedge \bar{Y} =_{\bar{V}_g} \bar{b}$$

To maintain V, insert all tuples $(\bar{a}',\bar{b},\bar{z}')$ where $\bar{z}'_g$, the $\bar{Z}_g$ components of $\bar{z}'$, is obtained from the query

$$\{\bar{z}'_g \mid (\exists \bar{Y},\bar{Z}') V(-,\bar{Y},\bar{Z}') \wedge \bar{Y} =_{\bar{V}_g} \bar{b} \wedge \bar{Z}' =_{\bar{Z}_g} \bar{z}'_g\}$$

(2) If no group in S has both hidden join variables and exposed private variables, then V is self-maintainable under the insertion if and only if:

$$\tau_{\bar{b}} \wedge V(\bar{a}',\bar{b}',-)$$

and V is not affected by the insertion.

(3) Otherwise, V is not self-maintainable under the insertion.

$\square$

EXAMPLE 3.4.1 Consider a view V defined by

$$v(X,Y,T) \text{ :- } r(X,Y,T) \;\&\; s(X,Y,X) \;\&\; s(Y,1,Y) \;\&\; t(X,Z) \;\&\; u(T,Z)$$

and let us insert $r(1,2,3)$. Applying Case *(1)* of Theorem 3.4.1, V is self-maintainable if and only if $v(1,-,3) \wedge \tau_{1,2,3}$ holds, where $\tau_{1,2,3}$ is defined by the program

$$\begin{array}{rcl} r_{1,2,3} & :- & s(1,2,1)\ \&\ s(2,1,2) \\ s(X,Y,X) & :- & v(X,Y,T) \\ s(Y,1,Y) & :- & v(X,Y,T) \end{array}$$

and rewritten as $v(1,2,-) \wedge [v(2,1,-) \vee v(-,2,-)]$, which further simplifies to $v(1,2,-)$. In other words, the view self-maintainability test is $v(1,-,3) \wedge v(1,2,-)$. □

Thus, for the special class of conjunctive-query views with only exposed self-joins, the solutions to the view self-maintenance problem are unions of conjunctive queries, where the subgoals are independent from each other. These queries can be executed in time linear in the size of the view instance and even constant time if the appropriate indexes are maintained at the warehouse. Conjunctive-query views with only exposed self-joins are the largest subclass of CQ views we know how to handle efficiently. We will provide solutions to the general case of CQ views in Chapters 5 and 6, which are unfortunately much less efficient than the results of this chapter.

Chapter 4

Exploiting Functional Dependencies

In Chapter 3, we addressed the strict view self-maintenance problem, that is, the view self-maintenance problem where no base relations and no base dependencies are used. There, we avoided using the base relations because their access can be expensive. We may not have complete knowledge of the base relations, but the next-best form of knowledge that is often available for free is *integrity constraints* the relations satisfy. A type of constraints that is commonly found in database systems is *functional dependencies* (abbreviated FD's). The main questions we address in this chapter are whether the use of FD's helps in view self-maintenance and, if affirmative, how easily we can derive the solutions.

We begin with a simple example to demonstrate that the use of functional dependencies does affect view self-maintainability. The self-maintainability tests are shown in the example without full explanation, but will be more formally rederived in later sections. While this example conveys the salient points of the work, it does not reflect the full complexity of the problem of efficient view self-maintenance under general FD's.

EXAMPLE 4.0.2 In its new marketing strategy to promote customer loyalty, TMart, our large retail chain, uses a data warehouse to collect customer purchase information, drawing on external data sources that may be its own operational databases or may belong to outside information brokers. The following source relations are used:

- $sales(Customer, Item, Store)$ contains sale transactions collected from local branches.
- $cust(Customer, Area, Bankcard)$ contains information about customers' place of residence and credit cards they possess, and is provided by a credit bureau.
- $comp(Rival, Item, Area)$ indicates the presence of competing retailers in some geographic area together with the products they carry. This information resides in a customized database provided by an outside broker.

A view V, materialized at the warehouse, is defined by the query:

$$v(C,A,B,S,I,R) \quad :- \quad sales(C,I,S) \ \& \ cust(C,A,B) \ \& \ comp(R,I,A)$$

asking for customers who purchased some merchandise from the chain while the same merchandise can be bought from a competitor that has a presence in their residence area.

A new transaction $sales(cindy, igloo, springfield)$ is reported in. Since relations *cust* and *comp* can be accessed only for a fee, the question is whether V can be updated without using these relations at all, that is, whether V is *self-maintainable* (SM). Suppose for a moment that we are totally ignorant about these relations; that is, no dependencies are known to hold among them. In this case, the most general condition that guarantees V to be SM (ref. Section 3.2) is

$$v(cindy, -, -, -, igloo, -)$$

denoting the presence of some tuple in the view with the specific constants *cindy* and *igloo* in the C and I components respectively. Essentially, the presence of such tuple prevents the "adversary" from inventing a new area x that satisfies both $cust(cindy, x, -)$ and $comp(-, igloo, x)$, thus forcing V's update to depend on *cust* and *comp* by making it include tuple $(cindy, x, -, springfield, igloo, -)$.

Now suppose the data source guarantees that $Customer \rightarrow Area$ holds in *cust*. Intuitively, if we know *cindy*'s residence area, the adversary is no longer free to invent a different residence area for *cindy*, and the occurrence of both *cindy* and *igloo* in the same V tuple does not seem to be needed to guarantee V's self-maintainability. Indeed, consider this particular view instance:

$$V = \{ \ (cindy, a, b, s, ice, r) \ , \ (carl, a, b', s', igloo, r') \ \}$$

On the one hand, to update V, we need to include at least $(cindy, a, b, springfield, igloo, r')$ in the insertion, since both $cust(cindy, a, b)$ and $comp(r', igloo, a)$ can be inferred to hold. On the other hand, for any tuple $(cindy, A, B, springfield, igloo, R)$ to be in V's update, A had better be a, or else *cindy* would have had two different places of residence. Furthermore, B had better be b, or else $cust(cindy, a, B)$ would have hold, and V would have contained $(cindy, a, B, s, ice, r)$ prior to the update. Similarly, R is identified with r'. Hence, the required view updates can be determined exactly without knowing the exact content of the base relations, and condition $v(cindy, -, -, -, igloo, -)$ is no longer necessary for the view to be SM. By ignoring functional dependencies, we have missed opportunities for saving base data accesses that may be costly. When the FD is taken into consideration, condition $v(cindy, -, -, -, igloo, -)$ can be replaced by the weaker condition

$$(\exists A) \ v(cindy, A, -, -, -, -) \wedge v(-, A, -, -, igloo, -)$$

which turns out to be the most general condition for SM given this FD. □

Example 4.0.2 shows that even simple functional dependencies can affect view self-maintainability. Moreover, we note that self-maintainability conditions under FD's now involve joins, which do not appear when no FD's are considered. But how much more complex can the view self-maintenance solutions get under general functional dependencies?

In this chapter, we consider the view self-maintenance problem for a single view where:

- No base relations are used, but functional dependencies are given.
- The view to maintain is defined by a conjunctive query with *no self-joins* and *no projections*.
- Base updates are single insertions.

The rest of this chapter is organized as follows.

Section 4.1, ***View Self-Maintenance under FD's,*** makes the definition of view self-maintenance more precise, now that functional dependencies are taken in consideration. In particular, the notion of database consistency is extended and we show that the canonical database is still consistent under this new notion.

Section 4.2, ***Rectifying View Definitions,*** extends the notion of rectified representation of view definitions originally defined in Section 2.4. We show that under FD's, constants and variable repetitions within each subgoal of the view definition can be ignored with no loss of generality.

Section 4.3, ***Key Concepts,*** introduces two key concepts that will be used in solving the problem of view self-maintenance in the presence of functional dependencies: *well-founded DAG*, a graph abstraction that captures the effect of FD's on self-maintainability; and *subgoal partitioning*, a refinement of a similar concept introduced in Section 3.2, that takes FD's into consideration.

Section 4.4, ***Deriving Self-Maintainability Tests for Insertions,*** shows three situations where view self-maintainability under an insertion is guaranteed: two conditions for *Forced-Exclusion* and a condition for *Forced-Exposure.* The first two conditions guarantee that the view is not affected by the insertion. The third condition guarantees uniqueness of the view's new state and a maintenance expression is given. Most importantly, we show that together these three conditions completely characterize view self-maintainability.

Section 4.5, ***Summary,*** summarizes the results we obtain for view self-maintenance under insertions in the presence of functional dependencies and suggests possibilities for further simplifications of the solutions.

4.1 View Self-Maintenance under Functional Dependencies

To maintain a materialized view V, we are given:

- A query Q that defines V in term of some database D. Let us emphasize again that in this chapter, Q is a conjunctive query with the restrictions of *no self-joins* and *no projections.*
- The instance V of the view itself.
- A tuple t to be added to D.
- A set $\mathcal{F}$ of functional dependencies that the relations in D satisfy. We use $SAT(D, \mathcal{F})$ to denote this fact.

We further assume:

- D is consistent with both V and $\mathcal{F}$. This assumption extends the view realizability assumption introduced in Section 2.3.
- Not only $SAT(D, \mathcal{F})$ holds, but also the insertion does not violate any dependencies in $\mathcal{F}$. We write this assumption as $SAT(D \cup \{t\}, \mathcal{F})$.

In general, a database D is said to be consistent with V, t, and $\mathcal{F}$ if $Q(D) = V$ and $SAT(D \cup \{t\}, \mathcal{F})$.

Definition 4.1.1 (**Self-Maintainability in the Presence of FD's**) View V is said to be *self-maintainable* under the insertion of t if $Q(D \cup \{t\})$ does not depend on D, as long as D is consistent with V, t, and $\mathcal{F}$. More formally:

$$\begin{aligned}(\forall D_1, D_2)\ [\ \ & Q(D_1) = Q(D_2) = V \wedge SAT(D_1 \cup \{t\}, \mathcal{F}) \wedge SAT(D_2 \cup \{t\}, \mathcal{F}) \\ & \Rightarrow Q(D_1 \cup \{t\}) = Q(D_2 \cup \{t\})]\end{aligned}$$

□

Recall the notion of canonical database, $Q^{-1}(V)$, defined earlier in Section 2.2. Since Q is restricted in this chapter to a conjunctive query with no projections, the exact same definition can be used here since it does not depend on functional dependencies. In addition, the following property, which does relate to functional dependencies, holds for $Q^{-1}(V)$.

Theorem 4.1.1 *Let V be a view defined by a conjunctive query Q over some database D. Let $\mathcal{F}$ be a set of functional dependencies that hold in D. Let t be a tuple whose insertion to D does not violate $\mathcal{F}$. Assume Q has no projection, and let $Q^{-1}(V)$ be the canonical database. Then $Q^{-1}(V)$ is consistent with V, t, and $\mathcal{F}$.* □

Proof: The fact that $Q(Q^{-1}(V)) = V$ follows from Theorem 2.3.1. Furthermore $Q^{-1}(V)$ is contained in any database that is consistent with V, and D in particular. Since we assume $D \cup \{t\}$ satisfies $\mathcal{F}$, it follows that $\mathcal{F}$ holds for any subset of $D \cup \{t\}$, and $Q^{-1}(V) \cup \{t\}$ in particular. ■

The fact that $Q^{-1}(V)$ is consistent with V, t, and $\mathcal{F}$ will be used later in finding counterexamples for the completeness proof in Section 4.4. Its importance will not be apparent until Chapter 5, when we develop a general method to solve the view self-maintenance problem.

For views defined by conjunctive queries with no self-joins and no projections, self-maintainability can be made simpler if we can ignore certain functional dependencies. In fact, the following theorem states that FD's on the updated relation can be ignored safely. In this theorem, we assume that the view definition is rectified as defined in Section 2.4.

Theorem 4.1.2 *Let V be a view defined by a conjunctive query with no self-joins and no projections. Let $\mathcal{F}$ be a set of functional dependencies that hold in the base relations. Let t be a tuple to be inserted into some base relation R, and let $\mathcal{F}'$ be the dependencies in $\mathcal{F}$ over base relations other than R. Under the insertion of t, V is self-maintainable in the presence of $\mathcal{F}$ if and only if it is self-maintainable in the presence of $\mathcal{F}'$.* □

Proof: We give only an informal proof here. A formal proof can be found later, in Chapter 5. Intuitively, dependencies over R can only be used to exclude certain tuples from R, based on the inserted tuple which we assume does not violate the dependencies. But since an instance of V is given, this information on R cannot constrain the possible contents of the nonupdated relations, which are used to determine the required update to V. Therefore, knowing the dependencies over R does not help determining whether or not V is self-maintainable. ∎

Note that Theorem 4.1.2 remains valid if we lift the assumption that the view definition is rectified. In fact, the next section shows that rectifying the view definition does not affect generality. As a corollary, the definition of self-maintainability in Definition 4.1.1 is equivalent to the following:

$$(\forall D_1, D_2)\ [Q(D_1) = Q(D_2) = V \wedge SAT(D_1, \mathcal{F}) \wedge SAT(D_2, \mathcal{F}) \Rightarrow Q(D_1 \cup \{t\}) = Q(D_2 \cup \{t\})]$$

where we no longer require D_1 and D_2 to continue to satisfy $\mathcal{F}$ after the insertion.

Theorem 4.1.2 will be used later in this chapter to simplify the representation of functional dependencies in our problem.

4.2 Rectifying View Definitions

In Section 2.4, we showed that for conjunctive-query views with no self-joins, constants and variable repetitions within subgoals in the view definition play no role in the strict view self-maintenance problem. In this section, the same observation applies when functional dependencies are considered. This observation allows us to use the rectified representation for our view definition in the view self-maintenance problem.

We first extend the rectified representation introduced in Section 2.4 to take FD's into consideration. We then show that no generality is lost if we solve the view self-maintenance using this rectified representation.

Definition 4.2.1 (**Rectified Representation in the Presence of FD's**) Let view V be defined by conjunctive query $Q : H \text{ :- } G_1\ \&\ \ldots\ \&\ G_n$, where H is the head with predicate

v that uses variables $\bar{X}$, and for every $i = 1, \ldots, n$, G_i is a subgoal with predicate r_i that uses variables $\bar{X}_i$. Constant symbols may be used in Q, and within each literal in Q, variables may occur more than once. Let $\mathcal{F}$ be a set of functional dependencies that hold in relations $R_1, \ldots R_n$. The *rectified representation* of $(v, Q, \mathcal{F}, r_1, \ldots, r_n)$ is $(v', Q', \mathcal{F}', r'_1, \ldots, r'_n)$, where the view predicate v', the query Q', and the FD's $\mathcal{F}'$ are defined as follows:

- View instance V' is defined in terms of V by: $v'(\bar{X}) = H$.
- View predicate v' is defined by query $Q' : v'(\bar{X}) \coloneq r'_1(\bar{X}_1)\ \&\ \ldots\ \&\ r'_n(\bar{X}_n)$.
- The set $\mathcal{F}'$ of FD's that hold in relations $R'_1, \ldots R'_n$ is constructed as follows. For every subgoal g in Q, the following rules determine, for every FD $\alpha \to \beta$ [1] (in $\mathcal{F}$) on the predicate of g, the FD (in $\mathcal{F}'$) on the predicate for the corresponding g' in Q':

Case 1 If β is equated in g to a constant, or if β and some attribute in α are equated in g, ignore $\alpha \to \beta$.

Case 2 Otherwise, eliminate any attribute in α that is equated to a constant in g and combine any pair of attributes that are equated in g.

□

In the rectified representation defined above, it is important to note the following:

- The insertion of tuple t into R_i is represented by the insertion of a tuple t' into R'_i if G_i matches $r_i(t)$. In this case, t' consists of the constants in the bindings produced by the successful match.
- In query Q', although a variable occurs at most once *within* each subgoal, it may occur in more than one subgoal.

The following theorem states that view self-maintainability can be equivalently decided using the rectified representation.

Theorem 4.2.1 *Let V be a view defined by a conjunctive query Q with no self-joins and no projections. Let $\mathcal{F}$ be a set of functional dependencies that hold in the base relations. Let V', Q', and $\mathcal{F}'$ be the corresponding view, query, and functional dependencies in the rectified representation. Let t be an insertion into some base relation and let t' be the corresponding insertion in the rectified representation. Then V is self-maintainable under the insertion of t in the presence of $\mathcal{F}$ if and only if V' is self-maintainable under the insertion of t' in the presence of $\mathcal{F}'$.* □

Proof: Here, we extend the proof of Theorem 2.4.1 to take functional dependencies into consideration. We will be referring to this proof in the following.

[1]In the remainder of this chapter, we assume, without loss of generality, that any FD $\alpha \to \beta$ has single-attribute right hand side, i.e., β is a single attribute.

IF: In the proof of Theorem 2.4.1, we started with an instance $I = (I_1, \ldots, I_n)$ of the base relations $R_1, \ldots, R_n$ that is consistent with V and we used this instance to construct an instance $I' = (I'_1, \ldots, I'_n)$ of the base relations $R'_1, \ldots, R'_n$. We then showed that I' is consistent with V'.

Here, the additional fact we need to show is that if $I \cup \{t\}$ satisfies $\mathcal{F}$, then $I' \cup \{t'\}$ also satisfies $\mathcal{F}'$. To see why, consider a functional dependency $\alpha' \rightarrow \beta' \in \mathcal{F}'$, and let t'_1 and t'_2 be two tuples from the same relation in $I' \cup \{t'\}$. Tuples t'_1 and t'_2 derive from some tuples t_1 and t_2 respectively from $I \cup \{t\}$. Also, since only Case 2 (ref. Definition 4.2.1) can generate FD's for D', $\alpha' \rightarrow \beta'$ must derive from some FD $\alpha \rightarrow \beta'$ from $\mathcal{F}$. Assume that t'_1 and t'_2 agree over α'. It follows that t_1 and t_2 also agree over α. Since $I \cup \{t\}$ satisfies $\mathcal{F}$, t_1 and t_2 must agree over β'. Hence, t'_1 and t'_2 agree over β'.

ONLY-IF: Similarly, in the proof of Theorem 2.4.1, we started with an instance $I' = (I'_1, \ldots I'_n)$ of the base relations $R'_1, \ldots, R'_n$ that is consistent with V' and we used this instance to construct an instance $I = (I_1, \ldots I_n)$ of the base relations $R_1, \ldots, R_n$. We then showed that I is consistent with V.

Here, the additional fact we need to show is that if $I' \cup \{t'\}$ satisfies $\mathcal{F}'$, then $I \cup \{t\}$ also satisfies $\mathcal{F}$. For functional dependencies in $\mathcal{F}$ that fall into Case 2 (ref. Definition 4.2.1), the proof is analogous to the *IF* part above, and is not shown here. We are only left with the task of showing that $I \cup \{t\}$ also satisfies those dependencies that fall into Case 1. So, let t_1 and t_2 be two tuples from the same relation (call it R_k) in $I \cup \{t\}$, let $\alpha \rightarrow \beta$ be an FD over R_k, and let g be the corresponding subgoal in Q. For the subcase (in Case 1) where β is equated in g to a constant, all tuples from R_k, and t_1 and t_2 in particular, have that constant for their β components. For the subcase where β and some attribute in α are equated in g, if t_1 and t_2 agree over α, then they must also agree over β since they both satisfy g. ∎

Consequently, the query defining the view V to maintain will be represented in the remainder of this chapter exactly as in Chapter 3, that is:

$$Q : v(\bar{X}, \bar{Y}, \bar{Z}) \text{ :- } r(\bar{X}, \bar{Y}) \ \& \ S(\bar{Y}, \bar{Z}).$$

where r is the updated predicate and S a conjunction of subgoals with nonupdated predicates. Under the insertion of $r(\bar{a}, \bar{b})$, we are interested in evaluating the query $\{\bar{Z} : S(\bar{b}, \bar{Z})\}$ in order to determine the required insertions to the view.

4.3 Key Concepts

In this section, we introduce the key concepts that will be used in the next section to derive the solutions for the problem view self-maintenance in the presence of functional dependencies. These concepts assume a rectified representation for the view definition.

4.3.1 The Well-Founded DAG

Functional dependencies normally relate the attributes of a predicate. But since the subgoals in Q are rectified, we can think of the FD's as relating the variables in Q, by further ignoring which predicates the original FD's apply to. Also note that the FD's originating from r can be ignored, since they have no effect on view self-maintainability (following Theorem 4.1.2, which takes advantage of the no-self-join and no-projection restrictions on the view definition). The set of FD's on query variables can thus be represented by an AND/OR graph called the *dependency AND/OR graph*, constructed as follows:

- Each variable in Q is associated with a node in the graph.
- For each FD $\alpha \rightarrow \beta$ (where α represents a set of variables in Q and β a single variable), there is an arc from each variable in α to β. The arcs from the variables in α to β form a set of AND-arcs.

Of special interest are those connected AND subgraphs

- That are acyclic,
- That have a single sink node,
- All of whose source nodes correspond to updated variables, and
- All of whose interior nodes correspond to private variables.

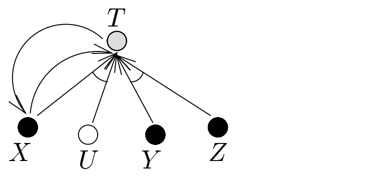
We call these subgraphs *well-founded derivation DAG's.* The single sink node of a well-founded derivation DAG is also called the *root.* With functional dependencies represented as such, the private variables are further categorized into determinable or nondeterminable. A private variable is said to be *determinable* when it corresponds to the root of some well-founded derivation DAG, *nondeterminable* otherwise.

EXAMPLE 4.3.1 Consider the view definition:

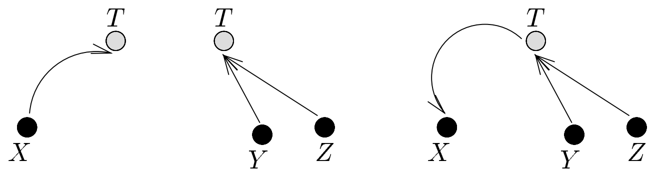
$$v(X, Y, Z, T, U) \quad :- \quad r(X, Y, Z) \ \& \ p(X, T, U) \ \& \ q(X, Y, Z, T).$$

and the FD's $XU \rightarrow T$ and $T \rightarrow X$ on p, and $X \rightarrow T$ and $YZ \rightarrow T$ on q. Using the notion of FD's on the view query variables, we simply say that X is determined by T, and T by either XU or YZ or X. The AND/OR graph that represents the dependencies between variables is depicted in Figure 4.1(a). Now, suppose r is the updated predicate. X, Y and Z are the updated variables, T and U private. All three well-founded derivation DAG's are shown in Figure 4.1(b), where the AND-connector that links an AND-node's incoming arcs is not shown. T is the only determinable variable, and U is nondeterminable. Updated nodes (nodes with an updated variable) are shown in Figure 4.1 in black, determinable nodes in grey, and nondeterminable nodes in white. □

Intuitively, determinable variables are those $\bar{Z}$-variables in query $S(\bar{Y}, \bar{Z})$ whose values are uniquely determined once the values of $\bar{Y}$ are fixed, provided that the tuples that "instantiate" certain dependencies are known to be present in the base relations. We say that the presence of these tuples *forces* the determinable variables in the query $S(\bar{Y}, \bar{Z})$ to agree on some specific values, making the query more specific. Also, we would like to point



(a) Dependency AND/OR graph



(b) Well-founded derivation DAG's

Figure 4.1: Graphs of dependencies between query variables.

out that there are only a finite number of well-founded derivations DAG's, and there are algorithms (e.g. depth first) to extract them all.

Since the FD's over the query variables are in fact functional dependencies that the view must satisfy, each well-founded derivation DAG corresponds to a chase (see [Ull89, AHV95]) of the view relation with these FD's that infers the functional dependence of a particular view attribute (corresponding to the DAG's root node) on a particular set of attributes (corresponding to the DAG's source nodes). One may wonder whether the complexity of a full DAG is really needed (since the DAG represents all the intermediate steps of the chase) and whether it is sufficient to keep only the source nodes and the root node (which correspond to the FD inferred by the chase). It turns out that as far as the view self-maintainability problem is concerned, it is generally not possible to abstract the DAG into just the source nodes and the root node, without losing completeness of the solution to the self-maintainability problem. This point will be substantiated in Example 4.4.5.

4.3.2 Generalizing the Subgoal Partitioning Concept

Recall from Chapter 3 the terminology we use to categorize the variables in the view definition:

- Variables in $\bar{X}$ and $\bar{Y}$ are called the *updated* variables.
- Variables in $\bar{Y}$ are called the *join* variables.
- Variables in $\bar{Z}$ are called the *private* variables.

In the presence of functional dependencies, we introduce the following notation that further refines the notion of private variables:

- Private variables $\bar{Z}$ will also be written as $\bar{D}, \bar{N}$, where
- $\bar{D}$ represents the determinable variables, and
- $\bar{N}$ represents the nondeterminable variables.

The partition operator $PART(-,-)$ was introduced in Section 3.2 to partition the subgoals with nonupdated predicates as follows:

$$PART(S(\bar{Y},\bar{Z}),\bar{Z}).$$

With functional dependencies, we still use the same operator but in the following *more refined partitioning* of the subgoals with nonupdated predicates:

$$PART(S(\bar{Y},\bar{D},\bar{N}),\bar{N}).$$

In other words, we now use the nondeterminable variables $\bar{N}$ as "glue", rather than all the private variables $\bar{Z}$.

EXAMPLE 4.3.2 Continuing from Example 4.3.1, with the given functional dependencies, U is the only nondeterminable variable. Thus, in contrast to the case with no dependencies, we consider the partitioning PART($\{p,q\}$,$\{U\}$) rather than PART($\{p,q\}$,$\{T,U\}$). The former is more refined than the latter since it consists of two groups ($\{p\}$ and $\{q\}$ in the former partitioning) instead of only one ($\{p,q\}$ in the latter partitioning). □

Intuitively, we will be looking for certain matching tuples in V that "conform" to the extended subgoal partitioning. This notion of "conform" will be made precise in a moment, but essentially the presence of such tuples assures that all required view updates can be computed from the view itself independently of the base relations.

4.4 Deriving Self-Maintainability Tests for Insertions

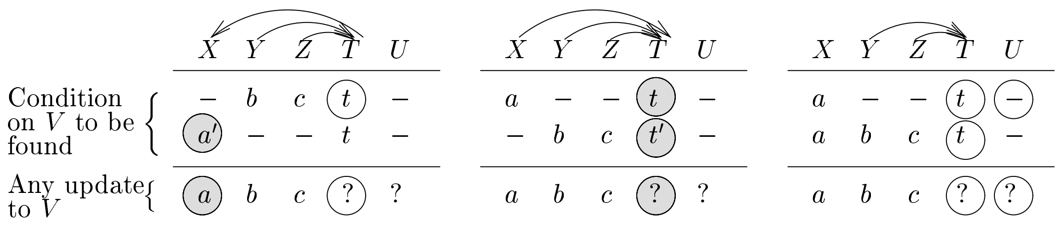
There are many ways a view can be self-maintainable under a given insertion. Perhaps the simplest is when the view is not affected by the insertion. So we begin looking for a condition on the view instance that guarantees no tuples in the base relations can join with the inserted tuple $r(\bar{a},\bar{b})$.

4.4.1 The Forced-Exclusion Conditions

Forced exclusion is a situation in which the presence of $S(\bar{b},\bar{Z})$, the tuples that join with $r(\bar{a},\bar{b})$, must be excluded in order to avoid conflicts due to the dependencies. The idea is to look for certain tuples in V that "instantiate" the dependencies in some well-founded derivation DAG.

EXAMPLE 4.4.1 Avoiding Conflicts over Updated Variables. Consider the view V and FD's in Example 4.3.1, and consider the insertion of $r(a,b,c)$. To update V on the one hand, any inserted tuple t_1 must be of the form (a,b,c,T,U). On the other hand, consider the set of dependencies $\{YZ \rightarrow T, T \rightarrow X\}$ that defines the well-founded derivation tree rooted at updated node X as shown in Figure 4.1(b), and suppose V contains some tuples $t_2 = (-,b,c,t,-)$ and $t_3 = (a',-,-,t,-)$ that "instantiate" these dependencies, where

$a' \neq a$. If the updated view contains t_1, then in chasing the dependencies in the derivation tree bottom up, $YZ \rightarrow T$ forces t_1 and t_2 to agree on $T = t$, which in turn leads $T \rightarrow X$ to force t_1 and t_3 to agree on $a = a'$, hence leading to a contradiction. Thus the presence of both t_2 and t_3 in V excludes the presence of t_1, or else a conflict would be created over updated variable X, as illustrated in Figure 4.2(a). □



(a) Conflict over X (b) Conflict over T (c) Exposure of T and U

Note: ● indicates conflict, ◯ indicates agreement.

Figure 4.2: Different conditions that guarantee V's updates to be independent of base relations.

Generalizing from Example 4.4.1, for each well-founded derivation DAG that is rooted at some updated node, we are looking for a set of tuples in V, one tuple for each dependency in the DAG, that instantiates the dependencies as follows: any tuple in the set agrees with the inserted tuple over the updated variables on the left hand side of its dependency; any pair of these tuples agrees over the determinable variables their dependencies may have in common; and the tuple for the root dependency disagrees with the inserted tuple over the root variable. Then $\mathcal{C}_{UPD}$, the disjunction of such conditions over all derivation DAG's rooted at some updated node, expresses the condition of forced exclusion that avoids conflicts over *updated* variables. The following formally defines $\mathcal{C}_{UPD}$.

Definition 4.4.1 (**Condition of forced exclusion based on avoiding conflicts over updated variables**) Consider the insertion of $r(\bar{a}, \bar{b})$. Let $\mathcal{D}$ a well-founded derivation DAG whose root node (the sink of the DAG) is a updated variable. Let $\alpha_{\mathcal{D}} \rightarrow \beta_{\mathcal{D}}$ be the FD at the root node, $\bar{D}_{\mathcal{D}}$ denote the variables at the internal nodes, and $DEP_{\mathcal{D}}$ denote the set of FD's at all $\mathcal{D}$'s internal nodes. We define $\mathcal{C}_{UPD}$ to be the following formula:

$$\bigvee_{\mathcal{D}} (\exists \bar{D}_{\mathcal{D}}) \; [\; (\exists \bar{X}, \bar{Y}, \bar{Z})(v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Z} =_{\alpha_{\mathcal{D}}} \bar{D}_{\mathcal{D}} \wedge \bar{Y} =_{\alpha_{\mathcal{D}}} \bar{b} \wedge \bar{Y} \neq_{\beta_{\mathcal{D}}} \bar{b})$$

$$\wedge \bigwedge_{\alpha \rightarrow \beta \in DEP_{\mathcal{D}}} (\exists \bar{X}, \bar{Y}, \bar{Z})(v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Z} =_{\alpha} \bar{D}_{\mathcal{D}} \wedge \bar{Y} =_{\alpha} \bar{b} \wedge \bar{Y} =_{\beta} \bar{b})]$$

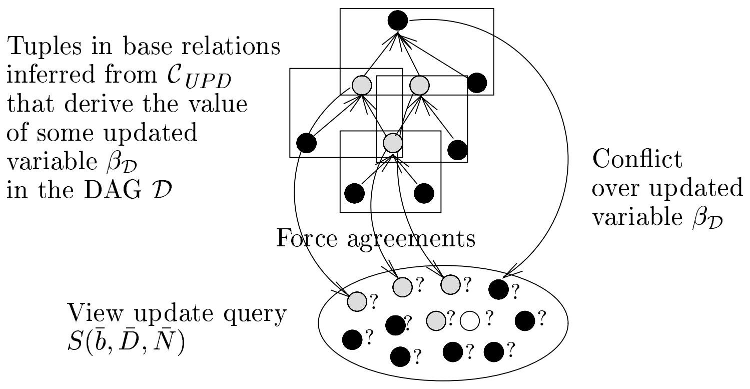

Tuples in base relations inferred from $\mathcal{C}_{UPD}$ that derive the value of some updated variable $\beta_{\mathcal{D}}$ in the DAG $\mathcal{D}$
Conflict over updated variable $\beta_{\mathcal{D}}$
Force agreements
View update query $S(\bar{b}, \bar{D}, \bar{N})$
?

Figure 4.3: A derivation-DAG view of computing the required view updates.

where the disjunction ranges over all DAG's with an updated root. □

Essentially, $\mathcal{C}_{UPD}$ looks for a well-founded derivation of a value for some updated variable that disagrees with $\bar{b}$ over that variable. We claim that when evaluated to true, this condition guarantees not only that the required update to the view does not depend on the base relations, but also that view does not require any update. The claim is stated in the following theorem.

Theorem 4.4.1 *Let V be a view, $\mathcal{F}$ a set of FD's over the base relations, and $r(\bar{a}, \bar{b})$ the inserted tuple. Let $\mathcal{C}_{UPD}$ be the condition as defined in Definition 4.4.1. If $\mathcal{C}_{UPD}$ holds, the insertion has no effect on the view.* □

Proof: Let us rewrite $\mathcal{C}_{UPD}$ as $\bigvee_{\mathcal{D}}(\exists \bar{D}_{\mathcal{D}})\theta_{\mathcal{D}}(\bar{D}_{\mathcal{D}})$. Intuitively, when there is a derivation DAG $\mathcal{D}$ and a constant value $\bar{d}_{\mathcal{D}}$ that satisfy $\theta_{\mathcal{D}}(\bar{D}_{\mathcal{D}})$, there are no other values that also satisfy the latter condition since all the nodes' values are uniquely determined by the source nodes of the $\mathcal{D}$. Furthermore, $\theta_{\mathcal{D}}(\bar{D}_{\mathcal{D}})$ assures the presence of certain tuples in the base relations that force the variables $\bar{D}_{\mathcal{D}} \subseteq \bar{D}$ in query $S(\bar{b}, \bar{D}, \bar{N})$ to agree with $\bar{d}_{\mathcal{D}}$ and also force disagreement at the root node, as depicted in Figure 4.3. Thus, the query $S(\bar{b}, \bar{D}, \bar{N})$ has no possible answer. ∎

There is yet another situation in which the view cannot gain any new tuples, but based on conflicts over determinable variables.

EXAMPLE 4.4.2 Avoiding Conflicts over Determinable Variables. Continuing from Example 4.4.1, now consider the tuples $t_4 = (a, -, -, t, -)$ and $t_5 = (-, b, c, t', -)$ where $t \neq t'$, that instantiate the dependencies defining the two derivation trees commonly rooted at determinable node T shown in Figure 4.1(b). If the updated view contains t_1, then in chasing the dependencies in both trees, t_1 and t_4 are forced to agree on $T = t$, and

t_1 and t_5 to agree on $T = t'$, leading to a contradiction again. Thus the presence of t_4 and t_5 in V also excludes the presence of t_1, but this time in order to avoid a conflict over determinable variable T (Figure 4.2(b)). □

Similarly, we can generalize from Example 4.4.2 to obtain $\mathcal{C}_{DET}$, the condition of forced exclusion that avoids conflicts over *determinable* variables, as follows: for each pair well-founded derivation DAG's that are commonly rooted at some determinable node, we look for tuples in V that instantiate the dependencies separately in each DAG, and such that the two tuples corresponding to the root dependencies disagree over the common root variable. The following formally defines $\mathcal{C}_{DET}$.

Definition 4.4.2 (**Condition of forced exclusion based on avoiding conflicts over determinable variables**) Consider the insertion of $r(\bar{a}, \bar{b})$. Let $\mathcal{D}_1$ and $\mathcal{D}_2$ two well-founded derivation DAG's whose root node is a determinable variable. Let $\alpha_{\mathcal{D}_1} \to \beta_{\mathcal{D}_1}$ be the FD at the root node of $\mathcal{D}_1$. Let $\bar{D}_{\mathcal{D}_1}$ denote the variables at the internal nodes of $\mathcal{D}_1$, and $DEP_{\mathcal{D}_2}$ the set of FD's at all internal nodes of $\mathcal{D}_1$. A similar notation is used for $\mathcal{D}_2$. We define $\mathcal{C}_{UPD}$ to be the following formula:

$$\begin{aligned}
\bigvee_{(\mathcal{D}_1,\mathcal{D}_2)} (\exists X_1, X_2)\ [\ & (\exists \bar{D}_{\mathcal{D}_1})[(\exists \bar{X}, \bar{Y}, \bar{Z})(v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Z} =_{\alpha_{\mathcal{D}_1}} \bar{D}_{\mathcal{D}_1} \wedge \bar{Y} =_{\alpha_{\mathcal{D}_1}} \bar{b} \wedge \bar{Z} =_{\beta_{\mathcal{D}_1}} X_1) \\
& \qquad \wedge \bigwedge_{\alpha \to \beta \in DEP_{\mathcal{D}_1}} (\exists \bar{X}, \bar{Y}, \bar{Z})(v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Z} =_{\alpha\beta} \bar{D}_{\mathcal{D}_1} \wedge \bar{Y} =_{\alpha} \bar{b})] \\
& \wedge (\exists \bar{D}_{\mathcal{D}_2})[(\exists \bar{X}, \bar{Y}, \bar{Z})(v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Z} =_{\alpha_{\mathcal{D}_2}} \bar{D}_{\mathcal{D}_2} \wedge \bar{Y} =_{\alpha_{\mathcal{D}_2}} \bar{b} \wedge \bar{Z} =_{\beta_{\mathcal{D}_2}} X_2) \\
& \qquad \wedge \bigwedge_{\alpha \to \beta \in DEP_{\mathcal{D}_2}} (\exists \bar{X}, \bar{Y}, \bar{Z})(v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Z} =_{\alpha\beta} \bar{D}_{\mathcal{D}_2} \wedge \bar{Y} =_{\alpha} \bar{b})] \\
& \wedge X_1 \neq X_2]
\end{aligned}$$

where the disjunction ranges over all pairs of DAG's sharing the same determinable root.
□

Essentially, $\mathcal{C}_{DET}$ looks for two different well-founded derivations for some determinable variable that yield different values. We claim that when evaluated to true, this condition guarantees not only that the required update to the view does not depend on the base relations, but also that view does not require any update. The claim is stated in the following theorem.

Theorem 4.4.2 *Let V be a view, $\mathcal{F}$ a set of FD's over the base relations, and $r(\bar{a}, \bar{b})$ the inserted tuple. Let $\mathcal{C}_{DET}$ be the condition as defined in Definition 4.4.2. If $\mathcal{C}_{DET}$ holds, the insertion has no effect on the view.* □

Proof: Intuitively, the condition assures the presence of certain tuples in the base relations that forces some determinable variable in the query $S(\bar{b}, \bar{D}, \bar{N})$ to agree with two conflicting values, each derived through a different derivation, as depicted in Figure 4.4. Hence, the query $S(\bar{b}, \bar{D}, \bar{N})$ has no possible answer. ∎

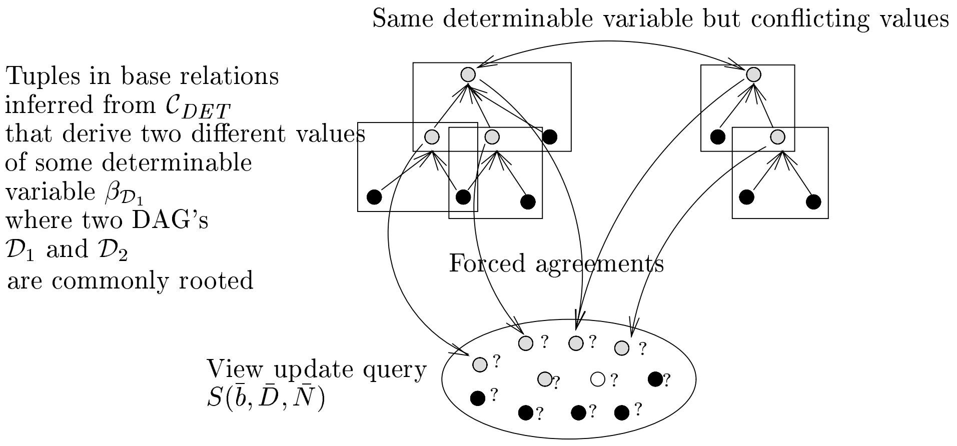

Same determinable variable but conflicting values
Tuples in base relations inferred from $\mathcal{C}_{DET}$ that derive two different values of some determinable variable $\beta_{\mathcal{D}_1}$ where two DAG's $\mathcal{D}_1$ and $\mathcal{D}_2$ are commonly rooted
Forced agreements
View update query $S(\bar{b}, \bar{D}, \bar{N})$
?

Figure 4.4: Two derivation DAG's disagreeing on their common root.

4.4.2 The Forced-Exposure Condition

We now turn to the case where view V may gain new tuples as a result of inserting $r(\bar{a}, \bar{b})$. *Forced exposure* is a situation in which the query $S(\bar{b}, \bar{Z})$ is forced to reveal the values for its private variables $\bar{Z}$ through the view, thus making the required updates to the view entirely computable from the view itself. Two key ideas come into play here. First, in the absence of FD's, the partition PART(S,$\bar{Z}$) allows us to generate the most general condition on V that forces all $\bar{Z}$-values in $S(\bar{b}, \bar{Z})$ to show up in the view (ref. Section 3.2), where PART(S,$\bar{Z}$) essentially treats $\bar{Y}$ as the only "bound" variables. Second, in the presence of FD's, this idea generalizes in a surprisingly simple way: treat all determinable variables in $\bar{Z}$ as bound, in addition to the $\bar{Y}$-variables. In other words, we consider the finer partition PART(S,$\bar{N}$), where $\bar{N}$ denotes the nondeterminable variables in $\bar{Z}$. Interestingly, the only aspect of FD's that counts is whether or not a variable is determinable.

EXAMPLE 4.4.3 Consider the view V and FD's in Example 4.3.1, and consider the insertion of $r(a, b, c)$. As in Example 4.4.1, any update to V is a tuple t_1 of the form (a, b, c, T, U). As shown in Example 4.3.2, PART($\{p(X, T, U), q(X, Y, Z, T)\}$,$\{U\}$) contains two groups $\{p\}$ and $\{q\}$. Corresponding to the first group, consider looking for some tuple in V that agrees with the inserted tuple over the group's updated variables, that is, $t_6 = v(a, -, -, t, -)$. Similarly for the second group, consider tuple $t_7 = v(a, b, c, t, -)$. Note that t_6 and t_7 are also required to agree over the determinable variable(s) their groups may share, T in this case. While the presence of either t_6 or t_7 in V forces t_1 to agree on $T = t$, their simultaneous presence assures that t_1's remaining unknown U can be determined by looking up $v(a, -, -, t, U)$ (see Figure 4.2(c)), independently of relations p and q: when $t_6 \in V$, $(a, t, U) \in p$ if and only if $(a, -, -, t, U) \in V$, and when $t_7 \in V$, $(a, b, c, t) \in q$. The alert reader may notice that in this case, the tuples to be inserted into V are already in the view, but this fact does not hold in general. □

Generalizing from Example 4.4.3, the presence of tuples that "conform" to PART(S,$\bar{N}$), such as t_6 and t_7 in Example 4.4.3, can be formalized in the following definition.

Definition 4.4.3 (**Condition of Forced Exposure**) Consider the insertion of $r(\bar{a}, \bar{b})$. The condition of *forced exposure*, $\mathcal{C}_A$, is defined by the following formula:

$$(\exists \bar{d}) \bigwedge_{g \in PART(S(\bar{Y},\bar{D},\bar{N}),\bar{N})} (\exists \bar{Y}, \bar{Z})\ v(-, \bar{Y}, \bar{Z}) \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \wedge \bar{Z} =_{\bar{D}_g} \bar{d}$$

where $\bar{Y}_g$ (resp. $\bar{D}_g$) denotes the updated (resp. determinable) variables used in group g, and $\bar{d}$ a vector of constants with the same dimension as $\bar{D}$. The fact that two vectors of constants $\bar{u}$ and $\bar{v}$ agree over variables $\bar{W}$ is denoted by $\bar{u} =_{\bar{W}} \bar{v}$. □

The following theorem shows that the condition of forced exposure is another way to guarantee view self-maintainability. Note how this theorem generalizes Theorem 3.2.1 for the case without functional dependencies.

Theorem 4.4.3 *Let $\mathcal{C}_A$ be the condition of forced exposure as defined in Definition 4.4.3 for a view V, a set of FD's, and the insertion of $r(\bar{a}, \bar{b})$. If $\mathcal{C}_A$ is satisfied (say with constant $\bar{d}$), then the required updates to V exactly consist of inserting all tuples $(\bar{a}, \bar{b}, \bar{d}, \bar{n})$ such that the $\bar{n}_g$ component of $\bar{n}$ is determined by*

$$\{\bar{n}_g \mid (\exists \bar{Y}, \bar{Z})\ v(-, \bar{Y}, \bar{Z}) \wedge \bar{Y} =_{\bar{Y}_g} \bar{a} \wedge \bar{Z} =_{\bar{D}_g} \bar{d} \wedge \bar{Z} =_{\bar{N}_g} \bar{n}_g\}.$$

□

Proof: First, when $\mathcal{C}_A$ is satisfied with some constant value $\bar{d}$ for variable $\bar{D}$, there are no other values that would satisfy it, since any value for $\bar{D}$ is uniquely determined by $\bar{b}$. This claim can be substantiated by considering every derivation DAG rooted at some determinable node and by showing that all interior nodes in the DAG are uniquely determined (starting from the deepest node and ending at the root node).

Furthermore, $\mathcal{C}_A$ assures the presence of certain tuples in the base relations that forces $\bar{D}$ in query $S(\bar{b}, \bar{D}, \bar{N})$ to agree with $\bar{d}$. This statement can be shown using arguments similar to showing the uniqueness of $\bar{d}$. The situation is illustrated in Figure 4.5.

So let $\bar{d}$ be the unique value that satisfies $\mathcal{C}_A$. We show that the query $\mathcal{P}(\bar{N})$: $\{\bar{N} \mid S(\bar{b}, \bar{d}, \bar{N})\}$, which determines the required updates to the view, can be computed from the view alone, regardless of the base relations. Consider the partition PART($S(\bar{Y}, \bar{D}, \bar{N})$,$\bar{N}$). Since the groups in the partition do not share any N-variables, $\mathcal{P}(\bar{N})$ is entirely determined if $S_g(\bar{b}_g, \bar{d}_g, \bar{N}_g)$ can be computed for every group g. Now we claim that

$$\{\bar{n}_g \mid S_g(\bar{b}_g, \bar{d}_g, \bar{n}_g)\} = \{\bar{n}_g \mid (\exists \bar{X}, \bar{Y}, \bar{Z})\ v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \wedge \bar{Z} =_{\bar{D}_g} \bar{d} \wedge \bar{Z} =_{\bar{N}_g} \bar{n}_g\}$$

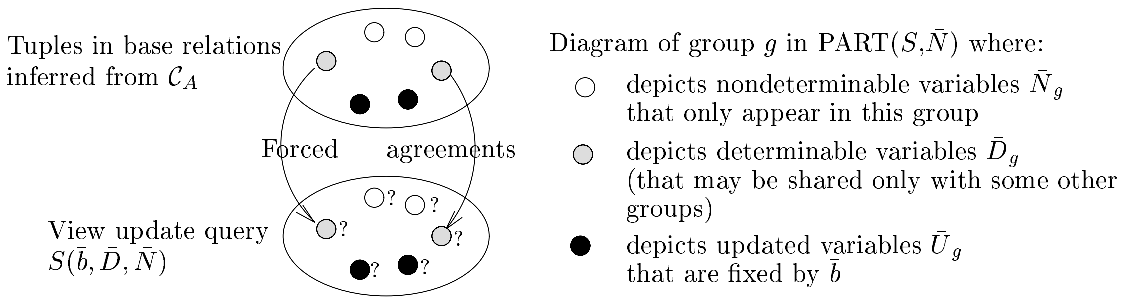

Tuples in base relations inferred from $\mathcal{C}_A$
Forced agreements
View update query $S(\bar{b}, \bar{D}, \bar{N})$
?
Diagram of group g in PART(S,$\bar{N}$) where:
○ depicts nondeterminable variables $\bar{N}_g$ that only appear in this group
◎ depicts determinable variables $\bar{D}_g$ (that may be shared only with some other groups)
● depicts updated variables $\bar{U}_g$ that are fixed by $\bar{b}$

Figure 4.5: A subgoal-partition view of computing required view updates.

The "$\supseteq$" part of the claim is obvious since the only way to explain the presence in V of a tuple that agrees with $\bar{b}$, $\bar{d}$, and $\bar{n}_g$ over variables $\bar{Y}_g$, $\bar{D}_g$, and $\bar{N}_g$ respectively, is the necessary presence of all the atoms in $S_g(\bar{b}_g, \bar{d}_g, \bar{n}_g)$.

For the "$\subseteq$" part of the claim, the satisfaction of $(\exists \bar{X}, \bar{Y}, \bar{Z}) v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Y} =_{\bar{Y}_g} \bar{b} \wedge \bar{Z} =_{\bar{D}_g} \bar{d}$ assures the presence of a set of tuples (one in each base relation not used in S_g) that join with $S_g(\bar{b}_g, \bar{d}_g, \bar{n}_g)$, for any $\bar{n}_g$. The result of this join will appear among the tuples in the view that agree with $\bar{b}$, $\bar{d}$, and $\bar{n}_g$ over variables $\bar{Y}_g$, $\bar{D}_g$, and $\bar{N}_g$ respectively. In other words, any $\bar{n}_g$ that satisfies $S_g(\bar{b}_g, \bar{d}_g, \bar{n}_g)$ will appear in the set on the right hand of the equality we are trying to prove. ∎

4.4.3 Complete Characterization of Self-Maintainability

Each of the three conditions $\mathcal{C}_A$, $\mathcal{C}_{UPD}$ and $\mathcal{C}_{DET}$ guarantees view self-maintainability. But are there other conditions that also provide that guarantee? In the following theorem, we claim that together, these three conditions completely characterize self-maintainability.

Theorem 4.4.4 *Let $\mathcal{C}_{UPD}$ and $\mathcal{C}_{DET}$ be the two conditions of forced exclusion, and $\mathcal{C}_A$ the condition of forced exposure as defined above for a given view, a given set of FD's and a given insertion. The view is self-maintainable under the insertion if and only if ($\mathcal{C}_{UPD} \vee \mathcal{C}_{DET} \vee \mathcal{C}_A$).* □

Proof:

IF: This part follows from Theorems 4.4.1, 4.4.2 and 4.4.3.

ONLY-IF: Assume $\mathcal{C}_A$, $\mathcal{C}_{UPD}$ and $\mathcal{C}_{DET}$ are all false. We need to find a counterexample that consists of two valid databases D_1 and D_2 that derive different views after the insertion.

Essentially, we choose D_1 to be $\mathcal{Q}^{-1}(V)$, and D_2 to be D_1 augmented with some set of tuples Δ constructed as follows. Δ initially includes some selected subset of $S(\bar{b}, \bar{D}, \bar{N})$, the choice being based on how $\mathcal{C}_A$ is falsified. The tuples in Δ may include variables. In chasing the FD's over D_2, some of these variables may be bound. The falsity of both $\mathcal{C}_{UPD}$ and

$\mathcal{C}_{DET}$ assures that the chase process terminates without encountering a contradiction. Any variable that remains unbound is replaced with some new constant, and some selected tuples are removed from Δ. The falsity of $\mathcal{C}_A$ assures that D_2 is valid and that $Q(D_2 \cup \{r(\bar{a}, \bar{b})\}) \neq \mathcal{Q}(D_1 \cup \{r(\bar{a}, \bar{b})\})$.

More precisely, we will consider the partitioning of $S(\bar{Y}, \bar{D}, \bar{N})$ at various levels of granularity: the coarser partition PART(S,$\bar{D} \cup \bar{N}$), the finer partition PART(S,$\bar{N}$), and some partition in between to be specified later. The coarser partition decomposes S into groups $g_i : S_i(\bar{Y}_i, \bar{D}_i, \bar{N}_i)$. The finer partition can be alternatively viewed as decomposing each g_i into the smaller groups $g_{ij} : S_{ij}(\bar{Y}_{ij}, \bar{D}_{ij}, \bar{N}_{ij})$. In this notation, we write $\mathcal{C}_A$ as

$$\bigwedge_i (\exists \bar{d}_i) \bigwedge_j (\exists \bar{X}, \bar{Y}, \bar{Z})\ v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Y} =_{\bar{Y}_{ij}} \bar{b} \wedge \bar{Z} =_{\bar{D}_{ij}} \bar{d}_i$$

Note that $\mathcal{C}_A$, the condition that guarantees the presence of tuples that "conform" to PART(S,$\bar{N}$), subsumes the condition that guarantees the presence of tuples that "conform" to the coarser partition PART(S,$\bar{D} \cup \bar{N}$). The latter condition can be written as $\bigwedge_i (\exists \bar{X}, \bar{Y}, \bar{Z})\ v(\bar{X}, \bar{Y}, \bar{Z}) \wedge \bar{Y} =_{\bar{Y}_i} \bar{b}$.

We show the claim by contradiction. So assume that $\mathcal{C}_A$, $\mathcal{C}_{UPD}$, $\mathcal{C}_{DET}$ are false. We construct two databases D_1 and $D_2 = D_1 \cup \Delta$ that are both valid prior to the insertion $r(\bar{a}, \bar{b})$ (i.e., $\mathcal{Q}(D_1) = Q(D_2) = V$, and both D_1 and D_2 satisfy the given FD's), but that derive different views after the insertion (i.e., $Q(D_1 \cup r(\bar{a}, \bar{b})) \neq Q(D_2 \cup r(\bar{a}, \bar{b}))$).

D_1 is taken to be the canonical database $Q^{-1}(V)$ which is already known to be valid. Δ is constructed as follows:

1. Since $\mathcal{C}_A$, written as $\bigwedge_i (\exists \bar{D}_i) \varphi_i(\bar{D}_i)$ for shorthand, is false, there must be some i such that $(\exists \bar{D}_i) \varphi_i(\bar{D}_i)$ is false. Initially, for each such i, we include $S_i(\bar{b}_i, \bar{D}_i, \bar{N}_i)$ in Δ. In the rest of the construction, any mention of i will refer to these groups.

2. We use the FD's to chase $D_1 \cup \Delta$ until quiescence. Conflicts do not arise since both $\mathcal{C}_{UPD}$ and $\mathcal{C}_{DET}$ are false. After quiescence, $D_1 \cup \Delta$ essentially satisfies all the FD's.

3. During the chase, some variables in $\bar{D}_i$ (say $\bar{E}_i$) are bound (to say $\bar{e}_i$), some other (say $\bar{F}_i$) remain unbound. Consider the partition PART(S_i,$\bar{F}_i \cup \bar{N}_i$) into groups g_{ij}. Note that this partition is coarser than PART(S_i,$\bar{N}_i$) used in $\mathcal{C}_A$, but finer than PART(S,$\bar{D} \cup \bar{N}$). Let ψ_{ij} denote $(\exists \bar{Y}, \bar{Z})[v(\bar{Y}, \bar{Z}) \wedge \bar{Y} =_{\bar{Y}_{ij}} \bar{b} \wedge \bar{Z} =_{\bar{E}_{ij}} \bar{e}_i]$.

4. Remove from Δ all those $S_{ij}(\bar{b}_{ij}, \bar{e}_{ij}, \bar{F}_{ij}, \bar{N}_{ij})$ such that ψ_{ij} is true, and bind all remaining unbound variables (i.e. $\bar{F}_{ij}$ and $\bar{N}_{ij}$) to new constants.

We claim that $Q(D_2 \cup r(\bar{a}, \bar{b})) \neq \mathcal{Q}(D_1 \cup r(\bar{a}, \bar{b}))$. It is easy to see that $\mathcal{Q}(D_2 \cup r(\bar{a}, \bar{b}))$ contains some tuple $(\bar{a}, \bar{b}, -)$, by construction of Δ. If at least a variable remains unbound when the chase reached quiescence, its value (a new constant) will show up in some tuple $(\bar{a}, \bar{b}, -)$ from $Q(D_2 \cup r(\bar{a}, \bar{b}))$. This tuple $(\bar{a}, \bar{b}, -)$ cannot be in $\mathcal{Q}(D_1 \cup r(\bar{a}, \bar{b}))$ (because of the new constants). If all variables are bound by the chase, then necessarily both $\bar{F}_{ij}$ and

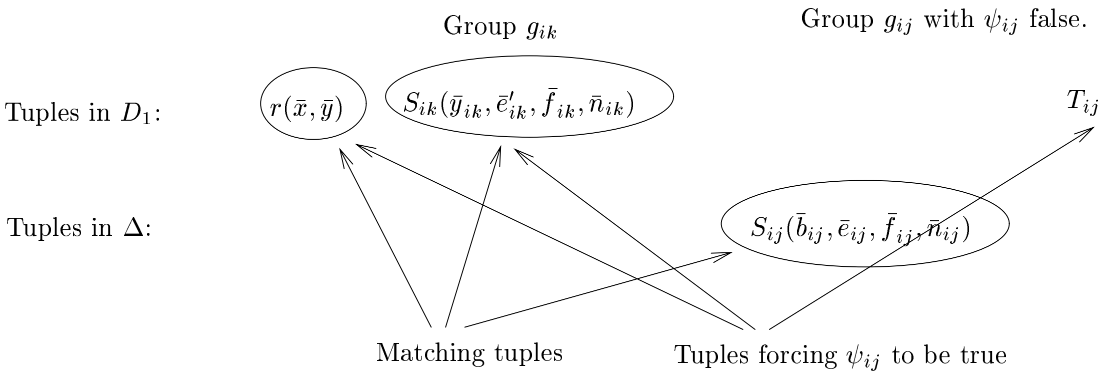

Group g_{ik}
Group g_{ij} with ψ_{ij} false.
Tuples in D_1:
$r(\bar{x}, \bar{y})$
$S_{ik}(\bar{y}_{ik}, \bar{e}'_{ik}, \bar{f}_{ik}, \bar{n}_{ik})$
T_{ij}
Tuples in Δ:
$S_{ij}(\bar{b}_{ij}, \bar{e}_{ij}, \bar{f}_{ij}, \bar{n}_{ij})$
Matching tuples
Tuples forcing ψ_{ij} to be true

Figure 4.6: Δ cannot contribute to the view.

$\bar{N}_{ij}$ are empty. Furthermore $Q(D_1 \cup r(\bar{a}, \bar{b}))$ cannot have any $(\bar{a}, \bar{b}, -)$ since otherwise, any $S_{ij}(\bar{b}_{ij}, \bar{e}_{ij})$ would have hold in D_1, contradicting the hypothesis that some ψ_{ij} is false.

We now claim that $Q(D_2) = V$. To prove our claim, we need to show that no tuples from Δ can contribute to the view (prior to insertion). Recall that only those groups g_{ij} such that ψ_{ij} is false do have tuples in Δ. If a Δ–tuple from a group g_{ij} contributes to the view, then all Δ–tuples from g_{ij} would also do so. Hence we can talk about group contribution instead of tuple contribution to the view. For each group g_{ij} such that ψ_{ij} is false, consider the set T_{ij} of D_1–tuples from S_{ij} that participate in the chase. Let $\{g_{ij} \mid j \in m_i\}$ be all the groups in the partition of S_i. Suppose the Δ–tuples from some of these groups that have ψ_{ij} false were to contribute to the view. Let n_i (a subset of m_i) denote these groups. Then D_1 must contain the following matching tuples:

- Tuples $S_{ik}(\bar{y}_{ik}, \bar{e}'_{ik}, \bar{f}_{ik}, \bar{n}_{ik})$ for all $k \in (m_i - n_i)$: $\bar{y}_{ik}$ agrees with $\bar{b}$ over $\bar{Y}_{ij}$ for $j \in n_i$, $\bar{e}'_{ik}$ agrees with $\bar{e}_i$ over $\bar{E}_{ij}$ for all $j \in n_i$, all remaining constants in $\bar{y}_{ik}$ and $\bar{e}'_{ik}$ agree across groups over the same variables.

- Tuple $r(\bar{a}, \bar{y})$: $\bar{y}$ agrees with $\bar{b}$ over $\bar{Y}_{ij}$ for all $j \in n_i$ and with all the $\bar{y}_{ik}$.

Since D_1 contains tuple $r(\bar{a}, \bar{y})$, the view must have some tuple $v(\bar{a}, \bar{y}, \bar{z})$. The tuples $S_{ik}(\bar{y}_{ik}, \bar{e}'_{ik}, \bar{f}_{ik}, \bar{n}_{ik})$ for all $k \in (m_i - n_i)$, together with T_{ij} for all $j \in n_i$, should force $\bar{z}$ to agree with $\bar{e}_i$ over $\bar{E}_{ij}$ for all $j \in n_i$. Hence, ψ_{ij} must hold for all $j \in n_i$, which contradicts the fact that all groups g_{ij}, $i \in n_i$, have ψ_{ij} false. In conclusion, no tuples from Δ can contribute to the view. ∎

In the following example, we apply Theorem 4.4.4 to find a complete self-maintainability test for the running example used in this chapter.

EXAMPLE 4.4.4 Consider the view and FD's in Example 4.3.1. A necessary and sufficient condition for the view to be SM under the insertion of $r(a, b, c)$ is obtained by combining the conditions that characterize the presence of t_2 and t_3, t_4 and t_5, t_6 and t_7 introduced in Examples 4.4.1, 4.4.2 and 4.4.3:

$$(\exists T)\ [v(-,b,c,T,-) \wedge v(\not a,-,-,T,-)] \vee (\exists T)\ [v(-,b,c,T,-) \wedge v(a,-,-,\not T,-)] \vee v(a,b,c,-,-)$$

where $\not a$ denotes any value but a. The first and last disjuncts combine to simplify to $v(-,b,c,-,-)$ which completely subsumes the second disjunct. Hence the view is SM if and only if $v(-,b,c,-,-)$. □

The following example illustrates a point made in Section 4.3 that we may loose generality if the well-founded derivation DAG is simplified into a representation that retains only the source nodes and the root node.

EXAMPLE 4.4.5 Consider the view definition

$$v(X,Y,Z,U) \coloneq r(X,Y,Z)\ \&\ s(Y,Z,U)\ \&\ t(X,U)$$

with the FD's $YU \to Z$ on s and $X \to U$ on t. Consider the self-maintainability of the view instance $V = \{(a,b',c,u),(a',b,c',u)\}$ under the insertion of $r(a,b,c)$. $\mathcal{C}_A$, expressed as $(\exists U)\ v(-,b,c,U) \wedge v(a,-,-,U)$, evaluates to false in the view instance. So the SM test degenerates to the conditions of forced exclusion. There are only two well-founded derivation DAG's, namely the trees represented by $\{YU \to Z, X \to U\}$ and $\{X \to U\}$ respectively. Thus, $\mathcal{C}_{DET}$ evaluates to false and $\mathcal{C}_{UPD}$ to true in our view instance, and the view is self-maintainable. However, had we collapsed the tree $\{YU \to Z, X \to U\}$ into the simpler tree $\{XY \to Z\}$, and used the latter tree to define $\mathcal{C}_{UPD}$, the simplified condition would have consisted of looking for tuples of the form $V(a,b,\not c,-)$. Obviously, this simplified condition evaluates to false in our view instance, showing that it is not necessary for self-maintainability. □

Finally, we apply Theorem 4.4.4 to rederive the self-maintainability test that was given without proof in Example 4.0.2 at the beginning of this chapter.

EXAMPLE 4.4.6 Consider the view defined in Example 4.0.2 with FD *Customer* $\to$ *Area*, and consider the insertion of *sales*(*cindy*, *igloo*, *springfield*). As there is only one derivation tree and its root A is determinable, the SM test degenerates to

$$(\exists A)\ v(\mathit{cindy},A,-,-,-,-) \wedge v(-,A,-,-,\mathit{igloo},-)$$

since both $\mathcal{C}_{UPD}$ and $\mathcal{C}_{DET}$ are vacuously false. □

4.5 Summary

In this chapter, we studied the problem of view self-maintenance under single insertions in the presence of functional dependencies. We considered the class of views defined by conjunctive queries with no self-joins and no projections.

We showed that view self-maintainability can be expressed by a boolean query that is a union of conjunctive queries with $\neq$ comparisons.

- The number of conjunctive queries in the union is bounded by
$$\prod_{\textit{Determinable variable } z} (\textit{number of FD's with } Z \textit{ on the right side})$$
Thus, for a given view definition, this number is a polynomial function of the number of functional dependencies.

- The number of conjuncts in the conjunctive queries by either the number of conjuncts or the number of determinable variables in the view definition. Thus, this number is linear in the size of the view definition.

In practice, there are additional steps we can take to minimize the solution complexity.

- The given functional dependencies may have redundancy among themselves. This redundancy can be removed using for example the technique for computing a *minimal cover*, as defined in [Ull89].

- There are techniques for optimizing union queries involving $\neq$ comparisons that can be applied to simplify our solution. For example, with any query P and constant $\bar{a}$:
$$P(\bar{a}) \vee (\exists \bar{X})\ [P(\bar{X}) \wedge \bar{X} \neq \bar{a}] \text{ simplifies to } (\exists \bar{X})\ P(\bar{X}).$$
And with any queries P and Q:
$$(\exists \bar{X})\ [P(\bar{X}) \wedge Q(\bar{X})] \vee (\exists \bar{X}, \bar{X}')\ [P(\bar{X}) \wedge Q(\bar{X}') \wedge \bar{X} \neq \bar{X}'] \text{ simplifies to}$$
$$(\exists \bar{X}, \bar{X}')\ [P(\bar{X}) \wedge Q(\bar{X}')].$$
These techniques can be used in conjunction with traditional techniques for minimizing conjunctive queries (see [Ull89]).

- It is easy to see that the dependencies over the variables in the view definition are in fact dependencies that hold in the view relation. Since the solutions are unions of conjunctive queries on the view relation, they can be optimized using known techniques for query optimization under dependencies. Such techniques can be found in [ASU79b, JK84, Sag87].

Finally, there are nontrivial special cases where the view self-maintenance admits very simple solutions. For example, when only key constraints are considered, if the updated relation does not join with any other relation on its key, then the key constraints play no role in view self-maintenance. To see why, consider the more general case where no functional dependency has all join variables on its left side. In this case, there are no well-founded derivation DAG's, and all private variables are nondeterminable. Therefore, this case degenerates to the case with no functional dependencies.

Chapter 5

Generalized VSM for Views with no Projections

The previous two chapters concentrate on the special cases of the view self-maintenance problem where:

- A single view is considered.
- The conjunctive query that defines the view has no self-joins.
- The types of updates allowed are restricted to updates to single relations, and to single insertions when functional dependencies are considered.

We took advantage of the restrictions on the problem to obtain solutions that are efficient and simple. The specialized methods we developed there are appealing both because they yield simple solutions and because they are direct and intuitive. For future work, it is therefore important to develop specialized methods that extend to more general cases or cover other special cases.

By contrast, in this chapter and the next, we emphasize on developing a general method that can be extended easily to cover different parts of the problem space. The following are the main motivating factors behind our desire to develop a general method:

- Developing specialized methods for different problem parameters often requires much effort and sometimes can be very difficult.
- There are problem parameters (such as the use of self-joins in the view definition) that, while simple on the surface, introduce complexity in the solution that is difficult to capture using specialized methods.

The following example illustrates the latter point.

EXAMPLE 5.0.1 Consider a view V defined by the following query

$$v(X, Y, Z) \coloneq r(X, Y) \ \& \ t(X, Z) \ \& \ t(Y, Z)$$

where predicate t is repeated in the body. Consider the insertion of $r(a,b)$. Consider for a moment a query obtained from the above by changing one of the occurrences of t to an unrelated predicate, say t'. Based on the results from Chapter 3, a self-maintainability condition for this modified query is given by:

$$v(a,b,-)$$

This condition, however, is not necessary for the original view to be self-maintainable. It turns out that the following condition is both sufficient and necessary for V's self-maintainability under the insertion of $r(a,b)$:

$$\begin{array}{l} v(a,b,-) \vee v(b,a,-) \vee \\ {[v(a,a,-) \wedge v(b,b,-)]} \vee \\ {[v(a,a,-) \wedge (\forall Z)(v(a,a,Z) \Rightarrow p_b(Z))]} \vee \\ {[v(b,b,-) \wedge (\forall Z)(v(b,b,Z) \Rightarrow p_a(Z))]} \end{array}$$

where $p_y(Z)$ is defined to be

$$\begin{array}{l} v(-,y,Z) \vee v(y,-,Z) \vee \\ (\exists X)[(v(X,y,-) \vee v(y,X,-)) \wedge \\ \qquad (v(X,-,Z) \vee v(-,X,Z))] \end{array}$$

This self-maintainability condition cannot be simplified much further. We have not found a method that is both intuitive and simple that can explain the complexity of this self-maintainability condition. □

In the remainder of this thesis, we will be using "generalized view self-maintenance" as a generic term to denote any part of the problem space that goes beyond what the two previous chapters already covered. This chapter considers the problem of generalized view self-maintenance with the following parameters:

- Multiple views are given.
- The views are defined by conjunctive queries with no projections (but with arbitrary self-joins).
- Functional dependencies over the base relations are given.
- A base update may be any mix of insertions and deletions.

For this problem, with the notable restriction of no projections in the views, we show how to derive a polynomial-time solution. This restriction will be lifted in the next chapter, but the solution will no longer be polynomial. The next chapter also shows how to extend the polynomial-time solution to cover the use of other additional information in maintenance (for views with no projections), and discusses other cases where the method leads to more complex solutions.

The rest of this chapter is organized as follows.

Section 5.1, *Generalized View Self-Maintenance,* gives a precise definition of the generalized view self-maintenance problem we consider in this chapter.

Section 5.2, *Canonical Databases,* extends the notion of database consistency and shows that, under this extended notion, canonical databases are consistent. Consistency of canonical databases is a key property underlying the general method we develop in this chapter.

Section 5.3, *Deriving Maintenance Expressions,* derives view maintenance expressions in the form of queries, based on canonical databases. The solution assumes that the given view is self-maintainable.

Section 5.4, *Deriving Self-Maintainability Tests,* reduces the self-maintainability problem to a problem of query containment. This reduction is based on canonical databases. While the query containment problem can be solved at runtime using known techniques, we take a step further: we show that we can translate the query containment problem to a boolean query in nonrecursive Datalog that decides self-maintainability at runtime.

5.1 Generalized View Self-Maintenance

Let $V_1, \ldots, V_m$ be the views in the warehouse, and let V_k be one of the view we would like to maintain. In addition to an instance of these views, we are given:

- For each $i = 1, \ldots, m$, a query Q_i that defines V_i in terms of some database D. D consists of the base relations $R_1, \ldots, R_n$. Each Q_i is a conjunctive query with no projections, but arbitrary self-joins are allowed.
- A set $\mathcal{F}$ of functional dependencies that holds in relations $R_1, \ldots, R_n$. We use $SAT(D, \mathcal{F})$ to denote this fact.
- An update U to database D that consists of δR_1^-, δR_1^+, δR_2^-, δR_2^+, $\ldots$, δR_n^-, δR_n^+, where δR_j^- (resp. δR_j^+) is the set of tuples to be deleted from (resp. inserted to) relation R_j. $U(D)$ denotes the updated database.

We assume that database D is consistent with views $V_1, \ldots, V_m$, update U, and $\mathcal{F}$, that is:

- D is consistent with $\mathcal{F}$ (i.e., $SAT(D, \mathcal{F})$) and with each V_i (i.e., $Q_i(D) = V_i$). This assumption extends the view realizability assumption introduced in Section 2.3.
- Update U does not violate any dependencies in $\mathcal{F}$. We write this assumption as $SAT(U(D), \mathcal{F})$.

We further assume:

$$\begin{tikzcd}
 & & & V_1 & & & \\
\mathcal{F} \arrow[r, dash] & D_1 \arrow[rru, "Q_1"] \arrow[rrd, "Q_m"'] \arrow[dd, "U"'] & & \vdots & & D_2 \arrow[llu, "Q_1"'] \arrow[lld, "Q_m"] \arrow[dd, "U"] \arrow[r, dash] & \mathcal{F} \\
 & & & V_m & & & \\
\mathcal{F} \arrow[r, dash] & U(D_1) \arrow[rr, "Q_k"] & & \overset{?}{=} & & U(D_2) \arrow[ll, "Q_k"'] \arrow[r, dash] & \mathcal{F}
\end{tikzcd}$$

Figure 5.1: Generalized view self-maintainability.

- Update U is meaningful, that is, the sets δR_j^- and δR_j^+ have no tuples in common for any j. In practice, any sequence of insertions and deletions can be represented as disjoint sets of insertions and deletions.

The following formalizes self-maintainability in the presence of the given information. Figure 5.1 shows how the parameters relate to each other in determining view self-maintainability.

Definition 5.1.1 (**Generalized Self-Maintainability**) Let $V_1, \ldots, V_m$ be views, and for $i = 1, \ldots, m$, let Q_i be the conjunctive query that defines V_i in terms of some database D. Let $\mathcal{F}$ be a set of FD's that hold in D. View V_k is said to be *self-maintainable* under a base update U if $Q(U(D))$ does not depend on D, as long as D is consistent with each V_i, U, and $\mathcal{F}$. More formally:

$$
\begin{aligned}
(\forall D_1, D_2)\ [\ & \bigwedge_i Q_i(D_1) = Q_i(D_2) = V_i \\
& \wedge\ SAT(D_1, \mathcal{F})\ \wedge\ SAT(D_2, \mathcal{F})\ \wedge\ SAT(U(D_1), \mathcal{F})\ \wedge\ SAT(U(D_2), \mathcal{F}) \\
& \Rightarrow Q(U(D_1)) = Q(U(D_2))]
\end{aligned}
$$

□

Thus, self-maintainability of V_k is a function of U and $V_1, \ldots, V_m$ (it is also a function of the view definitions Q_i and functional dependencies, but that is understood). Note the requirement that D be consistent with all the views, and not just with the view to maintain as in single-view self-maintainability.

When V_k is self-maintainable, the maintenance expression for V_k we are looking for is also a function of U and $V_1, \ldots, V_m$ (not just V_k as in single-view self-maintenance).

At first, the view self-maintainability condition as specified in Definition 5.1.1 does not appear to lend itself to a query containment formulation which typically uses only one quantified database variable instead of two. A key idea is to eliminate one of the two database variables by replacing it with some known database. This database, which

$v_1(X,Y,Z)$	$r(X,Y)$	$s(Y,Z)$	$t(Z)$
a_1,b_1,c_1	a_1,b_1	b_1,c_1	c_1
a_1,b_1,c_2	a_1,b_1	b_1,c_2	c_2

$v_2(Y,Z)$	$s(Y,Z)$
b_1,c_1	b_1,c_1
b,c_2	b,c_2
b_1,c_2	b_1,c_2

Figure 5.2: Two views and the associated canonical database.

will serve as a reference against which all consistent databases will be compared, must be consistent itself. Also, if we know such a reference database, we can use it to propagate the effects of the update to the view we want to maintain, if the view is self-maintainable under the update. But how can we find such a reference database? The answer lies in the canonical database.

5.2 Canonical Databases

Recall the definition of a canonical database given in Section 2.2 for the view self-maintainable problem that involves a single view. When several views are given, the definition can be extended by simply taking the union of all $Q_i^{-1}(V_i)$.

Definition 5.2.1 (**Canonical Database for Multiple Views**) Let $V_1, \ldots, V_m$ be views, and for $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) over relations $R_1, \ldots, R_n$ that defines V_i. The *canonical database*, denoted $\hat{D}$, consists of all the tuples obtained as follows: for each view V_i, every tuple in V_i that matches Q_i's head provides a substitution that grounds all the atoms in Q_i's body; include all these ground atoms in $\hat{D}$. More precisely, $\hat{D}$ is $\bigcup_i Q_i^{-1}(V_i)$, where $Q^{-1}(V)$ was defined in 2.2 for a view V defined by a CQ Q with no projection. $\Box$

EXAMPLE 5.2.1 Consider views V_1 and V_2 defined by:

$$v_1(X,Y,Z) \text{ :- } r(X,Y) \ \& \ s(Y,Z) \ \& \ t(Z)$$
$$v_2(Y,Z) \text{ :- } s(Y,Z)$$

Suppose $V_1 = \{(a_1,b_1,c_1),(a_1,b_1,c_2)\}$ and $V_2 = \{(b_1,c_1),(b,c_2),(b_1,c_2)\}$. The canonical database $\hat{D}$ in this view instance consists of $R = \{(a_1,b_1)\}$, $S = \{(b_1,c_1),(b_1,c_2),(b,c_2)\}$, and $T = \{(c_1),(c_2)\}$. The canonical database is shown in Figure 5.2 in table format. $\Box$

Intuitively, we are trying to reconstruct the base relations minimally from all the given views. When each Q_i has no projection, there is a unique minimal reconstruction, which is the canonical database $\hat{D}$. The following theorem states the key property of $\hat{D}$ that allows us to use it to maintain the views.

Theorem 5.2.1 *Let $V_1, \ldots, V_m$ be views, and for $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) over some database D that defines V_i. Let $\mathcal{F}$ be a set of functional dependencies that holds in D. Let U be an update to D that consists of arbitrary deletions and insertions and that does not violate $\mathcal{F}$. Then the canonical database $\hat{D}$ is consistent with $V_1, \ldots, V_m$, U, and $\mathcal{F}$.* □

Proof: For each $i = 1, \ldots, m$, the proof that $Q_i(\hat{D}) = V_i$ is analogous to the proof of Theorem 2.3.1. The proof that $\hat{D}$ satisfies $\mathcal{F}$ is analogous to the proof of Theorem 4.1.1: it follows from the facts that $\hat{D} \subseteq D$ and that D satisfies $\mathcal{F}$. Similarly, to show that $U(\hat{D})$ satisfies $\mathcal{F}$, we use the facts that $U(\hat{D}) \subseteq U(D)$ and that $U(D)$ satisfies $\mathcal{F}$. ■

5.3 Deriving Maintenance Expressions

In this section, we address the question of how to bring a view up to date *if the view is known to be self-maintainable.* Note that if a view is not self-maintainable, there is no unambiguous way to maintain the view correctly without using additional information.

We use a very simple idea. If a view is self-maintainable, we do not need to know what the actual database really is to maintain the view, since we can use any database that is consistent with all the views, the functional dependencies, and the given update, to propagate the update to the view. We can use the canonical database in particular. The following example illustrates how to maintain a view using the canonical database.

EXAMPLE 5.3.1 Continuing from Example 5.2.1, now consider inserting (a, b) to relation R. If V_1 is self-maintainable under the insertion (and with respect to the given view instance), we know we can obtain the same result for the new state of V_1 no matter which database we use to propagate the insertion, as long as it is consistent with the views, the functional dependencies, and the given update. We can use $\hat{D}$ in particular. So to compute the tuples gained by V_1, we simply join $r(a, b)$ with $S = \{(b_1, c_1), (b_1, c_2), (b, c_2)\}$ and $T = \{(c_1), (c_2)\}$ to obtain (a, b, c_2). □

The following theorem formalizes the use of $\hat{D}$ to maintain the views.

Theorem 5.3.1 *Let $V_1, \ldots, V_m$ be views, and for $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) over some database D that defines V_i. Let $\mathcal{F}$ be a set of functional dependencies that holds in D. Let U be an update to D that consists of arbitrary deletions and insertions and that does not violate $\mathcal{F}$. If view V_k is self-maintainable under U, then the new state for V_k is $Q_k(U(\hat{D}))$, where $\hat{D}$ is the canonical database.* □

Proof: The proof is illustrated is Figure 5.3. When view V_k is self-maintainable under update U, its new state is uniquely defined. Any database D that is consistent with $V_1, \ldots, V_m$, $\mathcal{F}$, and U will derive, after the update is made, the same view as $\hat{D}$. ■

$$\begin{tikzcd}
 & & V_1 & & \\
\mathcal{F} \arrow[r, dash] & D \arrow[ur, "Q_1"] \arrow[dr, "Q_m"'] \arrow[dd, "U"'] & \vdots & \hat{D} \arrow[ul, "Q_1"'] \arrow[dl, "Q_m"] \arrow[dd, "U"] \arrow[r, dash] & \mathcal{F} \\
 & & V_m & & \\
\mathcal{F} \arrow[r, dash] & U(D) \arrow[r, "Q_k"] & = & U(\hat{D}) \arrow[l, "Q_k"'] \arrow[r, dash] & \mathcal{F}
\end{tikzcd}$$

Figure 5.3: View maintenance with the canonical database.

Note that based on Theorem 5.3.1, $Q_k(U(\hat{D}))$, the new state of view V_k we want to compute, makes no reference to any functional dependencies. While functional dependencies are not directly part of the maintenance expression, they definitely affect self-maintainability of V_k. Also, note that since $\hat{D}$ is a function of (or more precisely, a simple query over) $V_1, \ldots, V_m$, the maintenance expression $Q_k(U(\hat{D}))$ is only a function of U and the V_i's. Finally, while Theorem 5.3.1 tells us how to compute the new state of V_k unambiguously, we actually do not want to recompute the entire view from scratch. In the following, we give an algorithm to compute the incremental maintenance expressions.

Algorithm 5.3.1 (Generate multiple-view self-maintenance queries)

Input: $Q_1, \ldots, Q_m$, where each Q_i is a conjunctive query (with no projection) that defines v_i using $r_1, \ldots, r_n$ as input, and is written as H_i :- G_{i1} & ... & G_{in_i}.

Output: Queries for incrementally maintaining V_k, using $v_1, \ldots, v_m$, $\delta r_1^-, \delta r_1^+, \ldots, \delta r_n^-, \delta r_n^+$ as input.

Method:

1. Generate the following rules that define the predicates $\hat{r}_1$,...,$\hat{r}_n$ for the canonical database, for $i = 1, \ldots, m$ and $j = 1, \ldots, n_i$:

   (A_{ij}) : $\hat{G}_{ij}$:- H_i

   where H_i is the head of Q_i and $\hat{G}_{ij}$ is the subgoal G_{ij} in Q_i's body whose predicate r_l is replaced by predicate $\hat{r}_l$.
2. Generate queries that incrementally maintain V_k, using predicates v_k, $r_1, \ldots, r_n$, $\delta r_1^-, \delta r_1^+, \ldots, \delta r_n^-, \delta r_n^+$ as input. Call this set of rules M.
3. Let $\hat{M}$ be obtained from M where every occurrence of r_j is replaced by $\hat{r}_j$, for $j = 1, \ldots, n$.
4. Return $\hat{M} \cup \{(A_{ij}), i = 1, \ldots, m, j = 1, \ldots n_i\}$.

▮

Step 1 in Algorithm 5.3.1 essentially computes the canonical database $\hat{D}$. Step 2 generates queries that incrementally maintain view V_k, i.e., that update V_k to the new state $Q_k(U(D))$ using V_k and all the base relations R_i's (the instance of these base relations is actually taken from the canonical database, which is the purpose of Step 3). Many known algorithms exist in the view-maintenance literature ([Kuc91, SJ96]) that can generate queries for incrementally maintaining a view using both the view and all the base relations, for example based on algebraic techniques for differentiating query expressions. Using for instance [SJ96] in Step 2, Algorithm 5.3.1 generates the queries that compute the required insertions to and deletions from a view, in time linear in the size of the view definitions. In practice, if these queries are optimized, we may not need to construct the entire canonical database as Step 1 would suggest.

EXAMPLE 5.3.2 Consider the view definitions for V_1 and V_2 from Example 5.2.1. Consider an update that consists of $\delta r^-, \delta r^+, \delta s^-, \delta s^+, \delta t^-, \delta t^+$. The order in which individual updates are applied to the base relations is immaterial, since we assume that for each relation, the set of tuples to be deleted is disjoint from the set of tuples to be inserted. Let δv_1^+ (resp. δv_1^-) be the predicate for the set of insertions (resp. deletions) that must be applied to to V_1. Step 1 of Algorithm 5.3.1 generates the following rules for the canonical database:

$\hat{r}(X,Y) \text{ :- } v_1(X,Y,Z)$
$\hat{s}(Y,Z) \text{ :- } v_1(X,Y,Z)$
$\hat{s}(Y,Z) \text{ :- } v_2(Y,Z)$
$\hat{t}(Z) \text{ :- } v_1(X,Y,Z)$

Using [SJ96] and exploiting the fact that V_1 is defined with no projection, Step 2 of Algorithm 5.3.1 generates the following rules for δv_1^+ and δv_1^-, using $\hat{r}$, $\hat{s}$, $\hat{t}$, δr^-, δr^+, δs^-, δs^+,δt^-, δt^+ as input (by differentiating the query expression that defines view V_1):

$\delta v_1^-(X,Y,Z) \text{ :- } \delta r^-(X,Y) \ \& \ \hat{s}(Y,Z) \ \& \ \hat{t}(Z)$
$\delta v_1^-(X,Y,Z) \text{ :- } \hat{r}(X,Y) \ \& \ \delta s^-(Y,Z) \ \& \ \hat{t}(Z)$
$\delta v_1^-(X,Y,Z) \text{ :- } \hat{r}(X,Y) \ \& \ \hat{s}(Y,Z) \ \& \ \delta t^-(Z)$
$\hat{r}_{new}(X,Y) \text{ :- } \hat{r}(X,Y) \ \& \ \neg\delta r^-(X,Y)$
$\hat{r}_{new}(X,Y) \text{ :- } \delta r^+(X,Y)$
$\hat{s}_{new}(Y,Z) \text{ :- } \hat{s}(Y,Z) \ \& \ \neg\delta s^-(Y,Z)$
$\hat{s}_{new}(Y,Z) \text{ :- } \delta s^+(Y,Z)$
$\hat{t}_{new}(Z) \text{ :- } \hat{t}(Z) \ \& \ \neg\delta t^-(Z)$
$\hat{t}_{new}(Z) \text{ :- } \delta t^+(Z)$
$\delta v_1^+(X,Y,Z) \text{ :- } \delta r^+(X,Y) \ \& \ \hat{s}_{new}(Y,Z) \ \& \ \hat{t}_{new}(Z)$
$\delta v_1^+(X,Y,Z) \text{ :- } \hat{r}_{new}(X,Y) \ \& \ \delta s^+(Y,Z) \ \& \ \hat{t}_{new}(Z)$
$\delta v_1^+(X,Y,Z) \text{ :- } \hat{r}_{new}(X,Y) \ \& \ \hat{s}_{new}(Y,Z) \ \& \ \delta t^+(Z)$

□

If a view is not self-maintainable, applying the maintenance queries generated by Algorithm 5.3.1 may incorrectly update the view. Thus, before applying them to maintain a view, it is important to make sure the view is self-maintainable. The next section provides a decision method.

5.4 Deriving Self-Maintainability Tests

As mentioned at the beginning of this chapter, up to now, we have solved the view self-maintainability problem only with the following parameters: single views with no self-joins and functional dependencies under single insertions. We do not yet have a solution for the problem that combines the following parameters: multiple views with self-joins, functional dependencies, and arbitrary mixes of base insertions and deletions. In this section, we develop a method that solves the view self-maintainability problem with such parameters.

To solve the self-maintainability problem, we reduce it to a problem of query containment that can be solved using known techniques. This reduction is based on the existence of a database we know how to build out of the contents of the views and that is consistent with all the views, functional dependencies, and the given update. Again, the canonical database can be used for this purpose. The following example illustrates this reduction.

EXAMPLE 5.4.1 Consider views V_1 and V_2 as defined in Example 5.2.1 and consider the insertion of $r(a, b)$. To determine whether view V_1 is self-maintainable under the insertion, the main idea is to compare the effect of the insertion on V_1 when using $\hat{D}$ with the effect when using any database consistent with both V_1 and V_2. First consider the view instance where $V_1 = \{(a_1, b_1, c_1), (a_1, b_1, c_2)\}$ and $V_2 = \{(b_1, c_1), (b, c_2), (b_1, c_2)\}$. V_1 is self-maintainable in this view instance because inserting $r(a, b)$ into any consistent database exactly causes (a, b, c_2) to be added to V_1, which is precisely the same effect on V_1 as the insertion into $\hat{D}$ (as determined in Example 5.3.1). Now consider another view instance where $V_1 = \{(a_1, b_1, c_1)\}$ and $V_2 = \{(b_1, c_1), (b, c_2)\}$. $\hat{D}$ in this case consists of $R = \{(a_1, b_1)\}$, $S = \{(b_1, c_1), (b, c_2)\}$, and $T = \{(c_1)\}$. V_1 is not self-maintainable in this view instance since while the insertion into $\hat{D}$ has no effect on V_1, there is a consistent database (namely $R = \{(a_1, b_1)\}$, $S = \{(b_1, c_1), (b, c_2)\}$, and $T = \{(c_1), (c_2)\}$) where the insertion of $r(a, b)$ causes V_1 to gain (a, b, c_2). □

5.4.1 Solving Self-Maintainability with Query Containment

Example 5.4.1 suggests that self-maintainability of a view under a given update can be characterized completely as the following implication problem: for every database D, if D is consistent with the views, the functional dependencies, and the given update, then D derives the same view as $\hat{D}$ after the update. This implication has the form of a query containment problem where the queries to compare are boolean queries. This reduction of self-maintainability to query containment is illustrated in Figure 5.4 and formalized in the following theorem.

$$\begin{tikzcd}
& & V_1 \\
\mathcal{F} \arrow[r, dash] & D \arrow[ur, "Q_1"] \arrow[dr, "Q_m"] \arrow[dd, "U"'] & \vdots \\
& & V_m \\
\mathcal{F} \arrow[r, dash] & U(D) \arrow[r, "Q_k"] & \overset{?}{=} Q_k(U(\hat{D}))
\end{tikzcd}$$

Figure 5.4: Reduction of view self-maintainability.

Theorem 5.4.1 *Let $V_1, \ldots, V_m$ be views, and for $i = 1, \ldots m$, let Q_i be the conjunctive query (with no projection) over some database D that defines view V_i. Let $\mathcal{F}$ be a set of functional dependencies that holds in D. Let U be an update to D that consists of arbitrary deletions and insertions and that does not violate $\mathcal{F}$. Let $\hat{D}$ be the canonical database. Then V_k is self-maintainable under U if and only if for every database D, $Q_k(U(D)) \neq Q_k(U(\hat{D}))$ implies $Q_i(D) \neq V_i$ for some i, $\neg SAT(D, \mathcal{F})$, or $\neg SAT(U(D), \mathcal{F})$.* $\square$

Proof:

IF: Assume $Q_k(U(D)) = Q_k(U(\hat{D}))$ holds for every database D that is consistent with all V_i's, $\mathcal{F}$, and U. Let D_1 and D_2 be two databases that are consistent with all V_i's, $\mathcal{F}$, and U. It follows that $Q_k(U(D_1)) = Q_k(U(\hat{D}))$ and $Q_k(U(D_2)) = Q_k(U(\hat{D}))$. We conclude $Q_k(U(D_1)) = Q_k(U(D_2))$, thus showing that V_k is self-maintainable under U.

ONLY-IF: Conversely, assume V_k is self-maintainable under U. Let D be a database that is consistent with all V_i's, $\mathcal{F}$, and U. Since $\hat{D}$ is also a database consistent with all V_i's, $\mathcal{F}$, and U (ref. Theorem 5.2.1), it follows that D and $\hat{D}$ derive the same view after update U is applied. ∎

Note that in Theorem 5.4.1, we use the implication "different-effect implies inconsistency" instead of "consistency implies same-effect". While both forms are equivalent, the queries to compare in the first one are slightly simpler. In the following, we use *DIFF* $\Rightarrow$ *INCON* or *DIFF* $\subseteq$ *INCON* to denote this implication. Theorem 5.4.1 allows us to solve the self-maintainability problem using known techniques for deciding whether a query is contained in another query. What is the nature of these queries? Let us write query Q_i, for each $i = 1, \ldots, m$, as

$$H_i :- G_{i1} \ \& \ \ldots \ \& \ G_{in_i}$$

Using this notation, queries *DIFF* and *INCON* from Theorem 5.4.1 can be elaborated. They are summarized in Table 5.1, where the rules correspond to the elements in Theorem 5.4.1 as follows:

- Rules (A_{ij}) compute $\hat{D}$, the canonical database.

Rules	Range	*DIFF*	*INCON*
(A_{ij})	$i = 1, \ldots, m$ $j = 1, \ldots, n_i$	$\hat{G}_{ij}$:- H_i	
(B_i)	$i = 1, \ldots, m$		*panic* :- G_{i1} & … & G_{in_i} & $\neg H_i$
(C_{ij})	$i = 1, \ldots, m$ $j = 1, \ldots, n_i$		*panic* :- H_i & $\neg G_{ij}$
(D_j)	$j = 1, \ldots, n$	r'_j :- r_j & $\neg \delta r^-_j$, r'_j :- δr^+_j	r'_j :- r_j & $\neg \delta r^-_j$, r'_j :- δr^+_j
(F_j)	$j = 1, \ldots, n$	$\hat{r}'_j$:- $\hat{r}_j$ & $\neg \delta r^-_j$, $\hat{r}'_j$:- δr^+_j	
(H_k)		$\hat{H}'_k$:- $\hat{G}'_{k1}$ & … & $\hat{G}'_{kn_k}$	
(I_k)		*panic* :- G'_{k1} & … & G'_{kn_k} & $\neg \hat{H}'_k$	
(J_{kj})	$j = 1, \ldots, n_k$	*panic* :- $\hat{H}'_k$ & $\neg G'_{kj}$	
$(L_{j\alpha\beta})$	$j = 1, \ldots, n$ FD $\alpha \to \beta$ on r_j		*panic* :- $r_j(\bar{X})$ & $r_j(\bar{X}')$ & $\bar{X} =_\alpha \bar{X}'$ & $\bar{X} \neq_\beta \bar{X}'$
$(M_{j\alpha\beta})$	$j = 1, \ldots, n$ FD $\alpha \to \beta$ on r_j		*panic* :- $r'_j(\bar{X})$ & $r'_j(\bar{X}')$ & $\bar{X} =_\alpha \bar{X}'$ & $\bar{X} \neq_\beta \bar{X}'$

Notation: H_i :- G_{i1} & … & G_{in_i} is the rule defining V_i; $\hat{G}_{ij}$ is the subgoal G_{ij} whose predicate r_l is replaced by predicate $\hat{r}_l$; $\hat{H}'_k$ is the head H_k whose predicate v_k is replaced by $\hat{v}'_k$; $\hat{G}'_{kj}$ is the subgoal G_{kj} whose predicate r_l is replaced by $\hat{r}'_l$; G'_{kj} is the subgoal G_{kj} whose predicate r_l is replaced by r'_l.

Table 5.1: Rules generated for queries whose containment decides self-maintainability of V_k.

- Rule (B_i) expresses the fact that $Q_i(D) \not\subseteq V_i$.
- Rules (C_{ij}) express the fact that $V_i \not\subseteq Q_i(D)$.
- Rule (D_j) defines predicate r'_j for relation R_j in $U(D)$.
- Rule (F_j) defines predicate $\hat{r}'_j$ for relation R_j in $U(\hat{D})$.
- Rule (H_k) defines predicate $\hat{v}'_k$ for $Q_k(U(\hat{D}))$, the new state of view V_k that derives from $\hat{D}$ after the update.
- Rule (I_k) expresses the fact that $Q_k(U(D)) \not\subseteq Q_k(U(\hat{D}))$.
- Rules (J_{kj}) express the fact that $Q_k(U(\hat{D})) \not\subseteq Q_k(U(D))$.
- Rule $(L_{j\alpha\beta})$ expresses the fact that relation R_j violates functional dependency $\alpha \to \beta$.
- Rule $(M_{j\alpha\beta})$ expresses the fact that relation $U(R_j)$ violates functional dependency $\alpha \to \beta$.

The following algorithm, based on Theorem 5.4.1, decides self-maintainability of a view under an arbitrary update and in the presence of other views and functional dependencies.

Algorithm 5.4.1 (Decide generalized view self-maintainability)

Input: Views $V_1, \ldots, V_m$, and for $i = 1, \ldots, m$, Q_i, a conjunctive query (with no projection) that defines v_i using $r_1, \ldots, r_n$ as input, a set $\mathcal{F}$ of functional dependencies satisfied by the r_i's, and an update $U = \delta R_1^-, \delta R_1^+, \ldots, \delta R_n^-, \delta R_n^+$. Query Q_i is written as H_i :- G_{i1} & ... & G_{in_i}.

Output: A decision whether V_k is self-maintainable under U in the given view instance $V_1, \ldots, V_m$.

Method:

1. Generate rules for the boolean queries *DIFF* and *INCON* as shown in Table 5.1. Both queries use the 0-ary predicate *panic* for their query predicate.
2. Return the decision whether *DIFF* $\subseteq$ *INCON* for every instance of $R_1, \ldots, R_n$.

∎

Referring to Table 5.1, each of *DIFF* and *INCON* is a Datalog query that can be transformed, after expanding rules (D_j) and eliminating all constant predicates, to a union of conjunctive queries. These queries involve negation and $\neq$ comparisons, and use $r_1, \ldots, r_n$ as input. Note that negation applies to these input predicates. The [LS93] algorithm can decide containment of unions of such queries in time exponential in the size of the views. As long as we use the reduction from Theorem 5.4.1, this complexity is probably the best that can be achieved, since the queries to compare use negation and use a number of constant symbols the size of the views. In the next section, we will give a more refined reduction that eliminates the use of negation, thus allowing more efficient containment checking algorithms to be used and, most importantly, self-maintainability to be decided in polynomial time.

5.4.2 Determining Self-Maintainability by Querying the View

Previously, we reduced self-maintainability to testing containment of queries, which involve negation that applies to the input predicates. This type of negation is the main source of complexity in the solution. We now show that this undesirable type of negation can be eliminated and that a more refined reduction can be obtained that results in simpler queries to compare.

Previously, self-maintainability of V_k under update U essentially reduces to checking whether after the update is made, every database D that is consistent with the views, dependencies, and update, derives the same relation as $Q_k(U(\hat{D}))$. The key observation here is that instead of checking all databases, we only need to check those that contain the canonical database $\hat{D}$. The simple reason is that any database that does not contain $\hat{D}$ cannot be consistent with the views, as formalized in the following lemma.

$$\begin{tikzcd}
 & & \subseteq V_1 \\
\mathcal{F} \arrow[r, dash] & D \cup \hat{D} \arrow[ur, "Q_1"] \arrow[dr, "Q_m"] \arrow[dd, "U"'] & \vdots \\
 & & \subseteq V_m \\
\mathcal{F} \arrow[r, dash] & U(D \cup \hat{D}) \arrow[r, "Q_k"] & \overset{?}{\subseteq} Q_k(U(\hat{D}))
\end{tikzcd}$$

Figure 5.5: A better reduction of view self-maintainability.

Lemma 5.4.1 *Let $V_1, \ldots, V_m$ be views, and for each $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) that defines V_i over some database D. If a database D is consistent with all the views, then $D \supseteq \hat{D}$, where $\hat{D}$ is the canonical database.* □

Proof: Let D be a database that is consistent with all the views. When each of the Q_i's is a conjunctive query without projection, each tuple in $\hat{D}$ is needed in order to explain the presence of some tuple in some view. In other words, every tuple in $\hat{D}$ must be present in D. ∎

The following example informally explains how checking self-maintainability can be improved.

EXAMPLE 5.4.2 Consider the same views as defined in Example 5.4.1, the view instance where $V_1 = \{(a_1, b_1, c_1), (a_1, b_1, c_2)\}$ and $V_2 = \{(b_1, c_1), (b, c_2), (b_1, c_2)\}$, and the insertion of $r(a, b)$. The new view V_1' that results from updating the canonical database has the additional tuple (a, b, c_2) besides those already in V_1. Previously, in order to determine if V_1 is self-maintainable under the insertion, we considered *every* database D and checked whether D exactly derives both V_1 and V_2 before the insertion and whether D exactly derives V_1' after. Improving upon the previous method, instead of considering all databases, now we consider only those that contain $\hat{D}$. Not only fewer databases need to be considered, their checks become considerably simpler, since a database that contains $\hat{D}$ must derive at least V_1 and V_2 before the update and at least V_1' after the update, and therefore these checks are not needed. □

The new reduction is depicted in Figure 5.5 and formalized in the following theorem, where $D \cup \hat{D}$ represents an arbitrary database that contains $\hat{D}$. The use of "set union" makes sense since a database is a set of tuples. Note that in order to represent a superset of $\hat{D}$, we do not use an arbitrary database D subject to the constraint $D \supseteq \hat{D}$, precisely because this constraint gives rise to the undesirable type of negation mentioned above.

Theorem 5.4.2 *Let $V_1, \ldots, V_m$ be views, and for $i = 1, \ldots, m$, let Q_i be the conjunctive query (with no projection) over some database D that defines V_i. Let $\mathcal{F}$ be a set of functional dependencies that holds in D. Let U be an update to D that consists of arbitrary deletions and insertions and that does not violate $\mathcal{F}$. Let $\hat{D}$ be the canonical database. Then V_k is self-maintainable under U if and only if for every database D, $Q_k(U(D \cup \hat{D})) \not\subseteq Q_k(U(\hat{D}))$ implies $Q_i(D \cup \hat{D}) \not\subseteq V_i$ for some i, $\neg SAT(D \cup \hat{D}, \mathcal{F})$, or $\neg SAT(U(D \cup \hat{D}), \mathcal{F})$. Furthermore, the boolean queries in the containment equation can be expanded to unions of conjunctive queries with negation that only applies to constant EDB predicates.* □

Proof: Following Theorem 5.4.1, V_k is self-maintainable under U if and only if $Q_k(U(D)) = Q_k(U(\hat{D}))$ for every database D that is consistent with the views, $\mathcal{F}$, and U. Applying Lemma 5.4.1, the validity of the latter statement does not change if we substitute "every superset of $\hat{D}$" for "every database D". Since a superset of $\hat{D}$ can be equivalently represented as $\hat{D} \cup D$ for some D, it follows that V_k is self-maintainable under U if and only if:

$$(\forall D) \bigwedge_i (Q_i(D \cup \hat{D}) = V_i) \wedge SAT(D \cup \hat{D}, \mathcal{F}) \wedge SAT(U(D \cup \hat{D}), \mathcal{F}) \Rightarrow Q_k(U(D \cup \hat{D})) = Q_k(U(\hat{D})) \tag{5.1}$$

Furthermore, for every i, since Q_i is monotonic and $Q_i(\hat{D}) = V_i$, $Q_i(D \cup \hat{D}) \supseteq V_i$ always holds. Thus, the $Q_i(D \cup \hat{D}) = V_i$ equality in (5.1) is equivalent to $Q_i(D \cup \hat{D}) \subseteq V_i$. Similarly, since both Q_k and U are monotonic, $Q_k(U(D \cup \hat{D})) \supseteq Q_k(U(\hat{D}))$ always holds. Thus, the last equality in (5.1) is equivalent to $Q_k(U(D \cup \hat{D})) \subseteq Q_k(U(\hat{D}))$.

To complete the proof, the boolean queries *DIFF* and *INCON*, whose containment decides self-maintainability of V_k, are listed in Table 5.2. In rules (B'_i) and (I_k), negation only applies to H_i and $\hat{H}'_k$, which are both defined entirely in terms of the constant predicates $v_1, \ldots, v_m$. Negation also appears when expanding the subgoals G'_{kj} in rule (I_k), but it only applies to the constant predicates δr_j^-. ∎

Table 5.2 lists the rules that define the boolean queries *DIFF* and *INCON*, whose containment decides self-maintainability of V_k in Theorem 5.4.2. The differences between this table and Table 5.1 can be highlighted as follows:

- Rules (C_{ij}) and (J_{kj}) from Table 5.1, which introduced negated subgoals with some variable predicate, have been eliminated.
- Rule (K_j) defines predicate r''_j for relation R_j in $D \cup \hat{D}$.
- Rule (B'_i) expresses the fact that $Q_i(D \cup \hat{D}) \not\subseteq V_i$. This rule replaces rule (B_i) from Table 5.1.
- Rule (D'_j) defines predicate r'_j for relation R_j in $U(D \cup \hat{D})$. This rule replaces rule (D_j) from Table 5.1.
- Rule (I_k) is unchanged from Table 5.1, but has a different meaning. It expresses the fact that $Q_k(U(D \cup \hat{D})) \not\subseteq Q_k(U(\hat{D}))$.

Rules	Range	*DIFF*	*INCON*
(A_{ij})	$i = 1, \ldots, m$ $j = 1, \ldots, n_i$	$\hat{G}_{ij}$:– H_i	$\hat{G}_{ij}$:– H_i
(K_j)	$j = 1, \ldots, n$	r''_j :– r_j, r''_j :– $\hat{r}_j$	r''_j :– r_j, r''_j :– $\hat{r}_j$
(B'_i)	$i = 1, \ldots, m$		*panic* :– G''_{i1} & … & G''_{in_i} & $\neg H_i$
(D'_j)	$j = 1, \ldots, n$	r'_j :– r''_j & $\neg\delta r^-_j$, r'_j :– δr^+_j	r'_j :– r''_j & $\neg\delta r^-_j$, r'_j :– δr^+_j
(F_j)	$j = 1, \ldots, n$	$\hat{r}'_j$:– $\hat{r}_j$ & $\neg\delta r^-_j$, $\hat{r}'_j$:– δr^+_j	
(H_k)		$\hat{H}'_k$:– $\hat{G}'_{k1}$ & … & $\hat{G}'_{kn_k}$	
(I_k)		*panic* :– G'_{k1} & … & G'_{kn_k} & $\neg\hat{H}'_k$	
$(L_{j\alpha\beta})$	$j = 1, \ldots, n$ FD $\alpha \to \beta$ on r_j		*panic* :– $r_j(\bar{X})$ & $r_j(\bar{X}')$ & $\bar{X} =_\alpha \bar{X}'$ & $\bar{X} \neq_\beta \bar{X}'$
$(M_{j\alpha\beta})$	$j = 1, \ldots, n$ FD $\alpha \to \beta$ on r_j		*panic* :– $r'_j(\bar{X})$ & $r'_j(\bar{X}')$ & $\bar{X} =_\alpha \bar{X}'$ & $\bar{X} \neq_\beta \bar{X}'$

Notation: $\hat{G}_{ij}$ is G_{ij} whose predicate r_l is replaced by $\hat{r}_l$; G''_{ij} is G_{ij} whose predicate r_l is replaced by r''_l; $\hat{H}'_k$ is H_k whose predicate v_k is replaced by $\hat{v}'_k$; $\hat{G}'_{kj}$ is G_{kj} whose predicate r_l is replaced by $\hat{r}'_l$; G'_{kj} is G_{kj} whose predicate r_l is replaced by r'_l.

Table 5.2: Rules generated for the queries to compare in the new reduction.

Theorem 5.4.2 improves on Theorem 5.4.1 in eliminating the uses of $\not\supseteq$ which were the main source of exponential complexity in Algorithm 5.4.1: $\not\supseteq$ introduced negation that applies to the variable EDB predicates $r_1, \ldots, r_n$. While negation still remains, it only applies to constant EDB predicates, which can be eliminated. In other words, the queries to compare are essentially unions of conjunctive queries with only $\neq$ comparisons, which are much simpler to deal with. We could have solved *DIFF* $\subseteq$ *INCON* directly by using known algorithms in the literature ([G*94, Klu88]) for deciding containment of unions of conjunctive queries with arithmetic comparisons. Even though these algorithms are more efficient than those for deciding containment of queries with negation, a naive way of applying them would require eliminating all constant EDB predicates. Unfortunately, the resulting complexity would still have been exponential in the size of the views, because the expanded queries have exponential size.

The next theorem is very important, since it gives us a *polynomial-time* solution to the self-maintainability problem. The key is to solve *DIFF* $\subseteq$ *INCON* without eliminating the constant EDB predicates and to translate it into a logical expression that involves these constant predicates rather than their extension. This expression is then rewritten as a query, which can be evaluated in time polynomial in the size of their extension.

Theorem 5.4.3 *Let DIFF and INCON be the two boolean queries in Theorem 5.4.2. There is a boolean query TEST that decides DIFF* $\subseteq$ *INCON. TEST is a nonrecursive Datalog query with negation and* $\neq$ *comparisons that only uses constant predicates as input.* □

Proof: Essentially, we first show that $DIFF \subseteq INCON$ can be characterized completely by a logical expression. We then show that this logical expression can be rewritten as a safe query. The full proof can be found in Appendix A. ∎

The following algorithm, based on Theorems 5.4.2, and 5.4.3, generates a boolean query that takes an instance of the views and an instance of the update as input, and determines whether or not a view is self-maintainable under an arbitrary update and in the presence of other views and functional dependencies.

Algorithm 5.4.2 (Generate generalized view self-maintainability test)

Input: $Q_1, \ldots, Q_m$, where each Q_i is a conjunctive query (with no projection) that defines v_i using $r_1, \ldots, r_n$ as input, and a set $\mathcal{F}$ of functional dependencies satisfied by the r_i's. Query Q_i is written as H_i :- G_{i1} & ... & G_{in_i}.

Output: A query that decides whether V_k is self-maintainable under U, using predicates $v_1, \ldots, v_m$ and $\delta r_1^-, \delta r_1^+, \ldots, \delta r_n^-, \delta r_n^+$ as input.

Method:

1. Generate rules for the boolean queries *DIFF* and *INCON* as shown in Table 5.2. Both queries use the 0-ary predicate *panic* for their query predicate.
2. Generate a query *TEST* that decides whether $DIFF \subseteq INCON$. Return *TEST*.

∎

More details on how Step 2 in Algorithm 5.4.2 can be implemented can be found in Appendix A.

Thus, in contrast to Algorithm 5.4.1 which decides self-maintainability at runtime, Algorithm 5.4.2 translates, at view-definition time, self-maintainability to a query test that can be evaluated against the views and the update instance at runtime. As such, not only can we test self-maintainability in polynomial time, but also we can optimize and compile the test more effectively than a test in procedural form such as Algorithm 5.4.1. The running time of Algorithm 5.4.2 and the size of the query test it generates do not depend on the instance of the views and update. They are exponential in the size of the view definitions. This complexity is not surprising, in light of the NP-completeness of checking query containment [CM77]. While the complexity of test generation is not as critical as the complexity of test execution, the availability of good query optimization techniques can help simplify the tests and further improve their execution speed.

EXAMPLE 5.4.3 Consider the definition of views V_1 and V_2 from Example 5.2.1 and consider the problem of testing self-maintainability of V_1 under the insertion of $r(a, b)$. Algorithm 5.4.2 generates a test which simplifies to the following query (using the 0-ary predicate *maintainable* as the query predicate):

$p(Z)$:- $v_1(X,Y,Z)$
$q(Z)$:- $v_2(Y,Z)$ & $v_1(X,Y,Z')$
depend :- $v_2(b,Z)$ & $\neg p(Z)$ & $\neg q(Z)$
maintainable :- $\neg$*depend*

□

5.5 Summary

In this chapter, we developed a general method for solving the generalized view self-maintenance problem in the presence of multiple views and functional dependencies, and under arbitrary sets of insertions and deletions, when the views do not have projections. We obtained the following results:

- When a view is self-maintainable, the problem of how to maintain the view without using any base relations can be reduced to a traditional view maintenance problem where the instance of all the base relations is available. The canonical database is used for such an instance.
- We reduced the view self-maintainability question to a question of query containment which we expressed as a boolean query against the views and the update instance. This query evaluates to *True* if and only if the view is self-maintainable. The query that tests self-maintainability is in nonrecursive Datalog with negation and $\neq$ comparisons. Thus, self-maintainability can be decided in time polynomial in the size of the view instance and the update instance. The size of the query itself is exponential in the size of the view definitions in the worst case. But since the query can be generated at compile-time, it can be optimized.

The method we developed in this chapter to solve the generalized view self-maintenance problem is based on concepts that are fairly general:

- Canonical databases.
- Reduction of self-maintainability to query containment.
- Expression of query containment as a query.

As will be shown in the next chapter, these concepts, the first two in particular, can be extended easily to handle additional parameters to the view self-maintenance problem that include tuple updates, partial copies (i.e., partial knowledge of the contents of a base relation), allowing projections in view definitions.

The general method can also serve as a tool that can help answer general questions that arise in the development of specialized methods for special cases. For example, we can easily show that functional dependencies on the updated relation play no role in determining self-maintainability of a view under insertions to a base relation, as stated in the following theorem.

Theorem 5.5.1 *Let V be a view defined by a conjunctive query Q (with no projections) over some database D. Let r be a base predicate that is not repeated in Q's body. Let $\mathcal{F}$ be a set of functional dependencies that holds in D. Let U be an update to D that consists of inserting a set δr of tuples to R (the relation for r) and that does not violate $\mathcal{F}$. Let $\mathcal{F}'$ be the dependencies in $\mathcal{F}$ over base relations other than R. Then under update U, V is self-maintainable in the presence of $\mathcal{F}$ if and only if it is self-maintainable in the presence of $\mathcal{F}'$.* □

Proof: As a special case of Theorem 5.4.2, V is self-maintainable under U if and only if for every database D,

$$Q(D \cup \hat{D}) \subseteq V \wedge SAT(D \cup \hat{D}, \mathcal{F}) \wedge SAT(U(D \cup \hat{D}), \mathcal{F}) \Rightarrow Q(U(D \cup \hat{D})) \subseteq Q(U(\hat{D})) \quad (5.2)$$

But $Q(U(D \cup \hat{D}))$ can be rewritten as $Q(D \cup \hat{D} \cup \delta R)$, which is the union of the following two relations:

- $Q(D \cup \hat{D})$, which is simply V, under the premise of (5.2).
- $Q_{\delta r}(D \cup \hat{D})$, where $Q_{\delta r}$ is the query obtained from Q by replacing r with predicate δr.

Thus, when we express (5.2) as the containment $\mathit{DIFF} \subseteq \mathit{INCON}$ in a way very similar to Table 5.2, predicate r does not appear in any rule in DIFF. Thus, any rule in INCON that uses predicate r can be dropped from consideration, since there is no containment mappings that map it to some rule in DIFF. In particular, any rule that expresses violation of R's dependencies can be dropped. In other words, self-maintainability of V under $\mathcal{F}$ and self-maintainability of V under $\mathcal{F}'$ are equivalent. ■

Theorem 4.1.2, which was stated in Section 4.1 without proof, is just a corollary of Theorem 5.5.1.

Chapter 6

Extensions

In the previous chapter, we developed a general method for solving a particular view self-maintenance problem. The purpose of this chapter is to demonstrate the extensibility of the method under other important view-maintenance situations.

In deriving the results in this chapter, we will reuse or extend the concepts presented in the previous chapter, and the following in particular:

- Canonical databases.

- Reduction of self-maintainability to query containment.

The rest of this chapter is organized as follows.

Section 6.1, *Dealing with Tuple Updates,* considers a kind of update that is very common in database systems, *tuple updates*. In general, a tuple update is *not* equivalent to a deletion followed by an insertion. We show a polynomial solution to this problem.

Section 6.2, *Using other additional information,* examines how we can exploit other common types of information such as a history of the most recent base updates, the assumption that all base updates are effective, and the use of a subset of the base relations. We show that under these situations, the view self-maintenance problem admits a polynomial solution.

Section 6.3, *Beyond CQ Views with no projections,* considers other classes of view definition that are important in practice: conjunctive queries with arithmetic comparisons, general conjunctive queries (with projection), and unions of conjunctive queries. For these classes of view, we give a solution to the view self-maintenance problem that is not polynomial.

Section 6.4, *Summary,* summarizes the results obtained by extending the method developed in the previous chapter and discusses future work in this area.

6.1 Dealing with Tuple Updates

So far we only consider base updates that consist of a set of tuples to be deleted from the base relations and a set of tuples to be inserted to these relations. However, we have not considered a kind of updates that is very common in database systems (exemplified by the SQL `update` command): *tuple updates* hereafter. A tuple update specifies two tuples t_{old} and t_{new}, and has the effect of replacing tuple t_{old} by tuple t_{new} if t_{old} is in the database. Note that in this update, the insertion of t_{new} is conditioned by the presence of t_{old} in the database. In other words, the update has no effect on the database if it does not already contain t_{old}.

By contrast, a base update that consists of the unconditional deletion of t_{old} and the unconditional insertion of t_{new} does not always have the same effect on the database. In particular, t_{new} is inserted into the database regardless of whether or not t_{old} was present. This difference raises the question as to whether or not a tuple update can be treated as a deletion and an insertion, as far as view self-maintainability is concerned. The following example confirms our hunch that a tuple update is generally not equivalent to a deletion and an insertion.

EXAMPLE 6.1.1 Consider a view V defined by

$$v(X,Y,Z) \coloneq r(X,Y) \ \& \ s(X) \ \& \ t(Y,Z)$$

Let U_1 consist of the deletion of $r(b,a)$ and the insertion of $r(a,b)$, and let U_2 consist of replacing $r(b,a)$ by $r(a,b)$. Consider the view instance $V = \{(a,y,z),(x,b,z')\}$. Applying Theorems 3.3.1 and 3.2.1, V is self-maintainable under U_1 because it contains some tuples of both forms $(a,-,-)$ and $(-,b,-)$. However, it is easy to show that V is not self-maintainable under U_2. Figure 6.1 shows two databases that are both consistent with the view instance prior to the update U_2, but that derive different views after the update. □

In the rest of this section, tuple updates on base relation $r(\bar{X})$ are represented by predicate $\gamma r(\bar{X},\bar{Y})$. The relation for γr contains a set of tuples $(\bar{x},\bar{y})$, each of which indicates a change of the R-tuple $(\bar{x})$ to the R-tuple $(\bar{y})$. Thus, using predicate $r'(\bar{X})$ for the updated relation, the effect of update $\gamma r(\bar{X},\bar{Y})$ on r can be represented by the following rules:

$$\begin{array}{rcl} dr(\bar{X}) & \coloneq & \gamma r(\bar{X},\bar{Y}) \\ r'(\bar{X}) & \coloneq & r(\bar{X}) \ \& \ \neg dr(\bar{X}) \\ r'(\bar{X}) & \coloneq & r(\bar{Y}) \ \& \ \gamma r(\bar{Y},\bar{X}) \end{array}$$

The following can be said about maintaining views under updates that also include tuple updates:

- The definition of canonical databases as in Definition 5.2.1 requires no change. Theorem 5.2.1, stating that the canonical database is consistent with the views, the update, and functional dependencies, remains valid. Theorem 5.3.1, the view maintenance theorem, remains valid.

$v(X, Y, Z)$	$r(X, Y)$	$s(X)$	$t(Y, Z)$
a, y, z	a, y	a	y, z
x, b, z'	x, b	x	b, z'
No changes.	U_2 has no effect on R		

Database D_1.

$v(X, Y, Z)$	$r(X, Y)$	$s(X)$	$t(Y, Z)$
a, y, z	a, y	a	y, z
x, b, z'	x, b	x	b, z'
	b, a		
a, b, z' added.	U_2 changes (b, a) to (a, b)		

Database D_2 derives differently from D_1 after the tuple update.

Figure 6.1: A non-self-maintainable view instance from Example 6.1.1.

.

- Theorem 5.4.2, the theorem that reduces self-maintainability to query containment, remains valid, since the update U is still a monotonic function of the database.
- Finally, although new conjunctive queries are added to each of *DIFF* and *INCON*, the resulting logical expressions continue to be in one of the forms specified in Table A.3. Therefore, Theorem 5.4.3, the theorem that states that $DIFF \subseteq INCON$ can be expressed as a safe query, remains valid.

6.2 Using other additional information

6.2.1 Using Partial Copies

So far, the base updates we considered in the view self-maintenance problem represent updates that have been *applied* to the base relations. We made no assumptions as to whether they represent updates that have been *effectively* applied, that is, whether they represent *net changes* to the base relations. For example, deletion of $\delta^- r$ and insertion of $\delta^+ r$ could have resulted in no change to relation R if no tuples of $\delta^- r$ are in R and every tuple of $\delta^+ r$ is in R. In practice, it is conceivable that the changes reported by the data sources represent net changes. In this case, a net deletion of $\delta^- r$ tells us that relation R must include certain tuples (namely, those specified by $\delta^- r$). Similarly, a net insertion of $\delta^+ r$ tells us that R must exclude other tuples (those specified by $\delta^+ r$).

The information that a base relation R include certain tuples and exclude others is called a *partial copy* of R. We call it a partial copy to contrast it with the case where the contents of R are completely known. Thus, given such additional information, the question is how we can take advantage of it in the view self-maintenance problem.

Another scenario where partial copies arise in practice is when the warehouse keeps a log of the most recent base updates. From this log, it is possible to infer some tuples that are not in the database and some of those that must be in the database.

For each base relation r, let r^+ represent a set of tuples that the relation for r must include and r^- a set of tuples that must be excluded. In the presence of this information, the following can be said about view self-maintenance:

- The definition of canonical databases from Definition 5.2.1 is extended to also include R_j^+ for each base relation R_j. In other words

  $$\hat{D} = \bigcup_i Q_i^{-1}(V_i) \cup \bigcup_j R_j^+$$

  Also, the notion of consistency is extended to include the requirement that a database include and exclude the given tuples. With these extended definitions, Theorem 5.2.1, stating that the canonical database is consistent with the views, the update, and functional dependencies, remains valid. Theorem 5.3.1, the view maintenance theorem, remains valid.

- Theorem 5.4.2, the theorem that reduces self-maintainability to query containment, is extended to include additional consistency conditions that database D excludes tuples from all R_j^-'s. In other words, view V_k is self-maintainable if and only if for every database D

  $$SAT(D \cup \hat{D}, \mathcal{F}) \wedge SAT(U(D \cup \hat{D}), \mathcal{F}) \wedge [\bigwedge_i Q_i(D \cup \hat{D}) \subseteq V_i] \wedge \bigwedge_j D \cap R_j^- = \emptyset$$

  implies

  $$Q_k(U(D \cup \hat{D})) \subseteq Q_k(U(\hat{D}))$$

  Note that since $\hat{D}$ is already known to exclude tuples from all the R_i^-'s, the additional consistency conditions are equivalent to the conditions that $D \cup \hat{D}$ excludes tuples from all R_j^-'s. Also, the condition that $D \cup \hat{D}$ includes all the R_j^+ is not needed since by definition, $\hat{D}$ already includes these R_j^+.

- In the resulting containment problem $DIFF \subseteq INCON$, we only need to include additional rules to $INCON$ that express $D \cap R_j^- \neq \emptyset$. Since these rules have a very simple form, namely

  $$panic :\!-\; r_j \;\&\; r_j^-,$$

  Theorem 5.4.3, the theorem that states that $DIFF \subseteq INCON$ can be expressed as a safe query, remains valid.

6.2.2 Using Base Relations

So far, we only considered strict self-maintenance, where we attempt to maintain the views using information that can be obtained strictly locally from the warehouse, namely the materialized views, the update, and possibly other information that is cheap to obtain. In particular, base access has been completely avoided.

But how do we proceed if a view turns out to be not self-maintainable in the strict sense? One possibility to fall back to the "normal" but expensive maintenance mode with unrestricted access to the base relations, as depicted in Figure 6.2(a). However, instead of switching to the normal maintenance mode immediately, another possibility is to use some (but not necessarily all) of the base relations to maintain the view. In fact, there are many cases where a view is not self-maintainable (in the strict sense) but can be maintained using some of the base relations. Thus, a more refined strategy based on full access to a subset of the base relations must be considered. Figure 6.2(b) illustrates this strategy. Note that the choice of which subset of base relations to use at each iteration is important not only because using different subsets incurs different costs, but also because different subsets provide different amounts of information relevant to self-maintainability. Thus, a subset that is expensive to use but that is likely to make a self-maintainability test succeed may be preferable over one that is cheap to use but that is unlikely to make the test succeed. In Figure 6.2(b), the choice of which subset of base relations to use next is left open. How to make the optimal choice is an important area for future research.

In the following, we show how to solve the generalized self-maintenance problem with full access to a specified subset of the base relations. There is a close resemblance between *allowing access to a base relation* and *having a copy of the base relation materialized at the warehouse*. In fact, if we assume that:

- The materialized views are *simultaneously* updated, that is, the required updates to each view are determined prior to updating any view,
- Update U is effective, that is, it represents the net changes to the underlying database, and
- The base relations are accessed in a state that reflects update U but no other later updates (assuming that the warehouse received updates in the order they are applied to the database),

then, the generalized self-maintenance problem can be treated as a strict self-maintenance problem where *a copy of the given base relations is available at the warehouse*, with the exception that the actual base relations and the copy only differ by the update.

EXAMPLE 6.2.1 Consider the problem of maintaining a view V defined by

$$v(X,Y,Z) :\!-\, r(X,Y) \;\&\; s(Y,Z) \;\&\; t(Z)$$

where we are allowed to access base relation S but not R or T. Consider an update with δR^-, δR^+, δS^-, δS^+, δT^-, and δT^+. The maintenance expression and maintainability

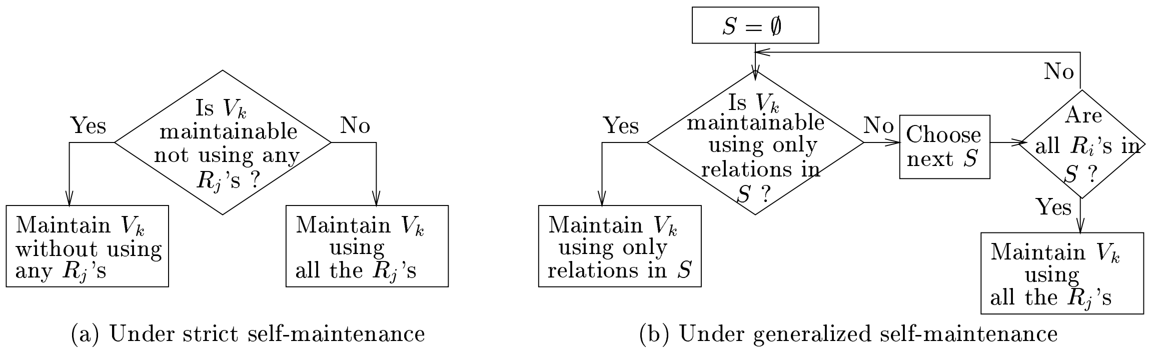

Is V_k maintainable not using any R_j's ?
Yes
No
Maintain V_k without using any R_j's
Maintain V_k using all the R_j's
(a) Under strict self-maintenance
$S = \emptyset$
Is V_k maintainable using only relations in S ?
Yes
No
Choose next S
Are all R_i's in S ?
No
Yes
Maintain V_k using only relations in S
Maintain V_k using all the R_j's
(b) Under generalized self-maintenance

Figure 6.2: Strategies for efficient warehouse maintenance.

test for this generalized self-maintenance problem can be obtained as follows. Consider the problem of maintaining V without accessing any base relation but where a copy of relation S is maintained at the warehouse as view V':

$$v'(Y, Z) :\!-\ s(Y, Z)$$

We can derive a solution to this problem that uses predicates v and v' as input. A solution to the original problem can be obtained from this solution by replacing every occurrence of v' with a new predicate s' defined by the following rules:

$$s'(Y, Z) :\!-\ s(Y, Z)\ \&\ \neg\delta s^{+}(Y, Z) \mid \delta s^{-}(Y, Z)$$

Predicate s' represents the state of relation S prior to the given update. □

Thus, results for the strict self-maintenance problem can be carried over by simply replacing every reference to the "copy" of a base relation by a reference to its "before image." In practice, allowing access to a base relation when maintaining a materialized view must be handled carefully. When a base relation is asynchronously updated by the source, it may be read by the warehouse in a different state than what is assumed by the warehouse. This situation may lead to erroneous updates to the warehouse, as reported in [Z*95]. Thus, a warehouse system that uses generalized self-maintenance must either allow access only to base relations that change in lock step with the warehouse, or combine our techniques with the compensation techniques developed in [Z*95].

6.3 Beyond CQ Views with no projections

6.3.1 CQ Views with Arithmetic Comparisons

Consider views defined by conjunctive queries with no projections but that allow arithmetic comparisons of the form (μ *op* ν), where *op* is one of the $<$, $\leq$, $>$, $\geq$, and $\neq$ operators, and μ and ν are either constants or variables.

The following can be said about maintaining CQ views with arithmetic comparisons:

- The definition of canonical databases as in Definition 5.2.1 requires no change. Theorem 5.2.1, stating that the canonical database is consistent with the views, the update, and functional dependencies, remains valid, since the queries defining the views are still monotonic. Theorem 5.3.1, the view maintenance theorem, remains valid.
- Theorem 5.4.2, the theorem that reduces self-maintainability to query containment, remains valid, since the update U is still a monotonic function of the database.
- The listings of queries *DIFF* and *INCON* as shown in Table 5.2 remain valid. These queries are unions of conjunctive queries with arithmetic comparisons, and exponential-time solutions exist in the literature ([G*94, Klu88]) for deciding $\mathit{DIFF} \subseteq \mathit{INCON}$. However, whether or not $\mathit{DIFF} \subseteq \mathit{INCON}$ can be expressed as a nonrecursive Datalog query is still open. It is also open as to whether or not $\mathit{DIFF} \subseteq \mathit{INCON}$ can be solved in time polynomial in the size of the input relations.

6.3.2 CQ Views with Projection

Consider views defined by conjunctive queries where some variables used in a rule's body do not appear in the rule's head. We call these variables hidden variables. For simplicity, we will first ignore functional dependencies. We will discuss how to extend the results to handle functional dependencies at the end of this subsection. A view where some attributes have been projected out looses information, and from an instance of the view, there is no unique way of minimally reconstructing the underlying database. The notion of canonical database from Section 5.2 must be revised to capture this nonuniqueness. The following redefines our notion of canonical database (note that the following definition is almost identical to Definition 2.2.1 with the only exception that a single view is assumed in the latter).

Definition 6.3.1 (**Canonical database for CQ views with projection**) Let $V_1, \ldots, V_m$ be views, and for $i = 1, \ldots, m$, let Q_i be the conjunctive query that defines V_i. The *canonical database*, denoted $\hat{D}$, consists of all the tuples obtained as follows: for each V_i and for each tuple t in V_i, the matching of t with Q_i's head provides a substitution for the variables in Q_i's body that also appear in the head; this substitution is extended to the remaining (hidden) variables by paring each of them to a *new symbol*; the ground atoms obtained after making this extended substitution into Q_i's body are included in $\hat{D}$. We will use $newsym(\hat{D}, t)$ to denote the new symbols generated for $\hat{D}$ due to tuple t, and $newsym(\hat{D})$ to denote all the new symbols generated for $\hat{D}$. □

EXAMPLE 6.3.1 Consider the view definition $v(X, Z)$:- $s(X, Y)$ & $s(Y, Z)$ where Y has been projected out. Consider the instance $V = \{(a_1, c_1), (a_2, c_2)\}$. The redefined canonical database $\hat{D}$ consists of $S = \{(a_1, y_1), (y_1, c_1), (a_2, y_2), (y_2, c_2)\}$, where y_1 and y_2 are new symbols. In this example, $newsym(\hat{D}, V(a_2, c_2)) = \{y_2\}$ and $newsym(\hat{D}) = \{y_1, y_2\}$ □

A tuple in $\hat{D}$ that contains a new symbol represents a fact involving some object whose value is not known. This value could be any of the known constants from the instance of the views or the update instance, or could be some constant not in any of those instances. Thus, if we consider all the symbol mappings h that map each of the new symbols to either one of themselves or a known constant, then $\hat{D}$ represents not a single database but a class of possible databases, each of which is obtained by applying some mapping h to $\hat{D}$. The following example illustrates the nonuniqueness of canonical databases due to projections in views.

EXAMPLE 6.3.2 Consider the same view definition as in Example 6.3.1, but a different view instance $V = \{(d, c)\}$. Consider the insertion of (a, b) into S. The canonical database $\hat{D}$, which consists of $S = \{(d, y), (y, c)\}$ where y is a new symbol, actually can be interpreted in five possible ways (by mapping y to either y, a, b, d, or c): $S = \{(d, y), (y, c)\}$, $S = \{(d, a), (a, c)\}$, $S = \{(d, b), (b, c)\}$, $S = \{(d, d), (d, c)\}$, or $S = \{(d, c), (c, c)\}$. The last two databases are not consistent with V, since they respectively derive tuples (d, d) and (c, c) which are not in the view. Among the remaining consistent databases, after the insertion, the second one derives tuple (d, b) not derived by the first one. Thus, view V is not self-maintainable under the insertion of (a, b) to S. □

A mapping that gives an interpretation of $\hat{D}$ that is consistent with all the views is said to be *consistent*. But do consistent mappings always exist? Before answering this question, let us first make this notion of mapping more precise, define the notion of isomorphic databases, and state two lemmas that will be useful to answer the question.

Definition 6.3.2 (**Canonical Mappings**) Given a set of views and together with their definitions, let K be a set of symbols that contains symbols used in the views and view definitions. Let $\hat{D}$ be the canonical database (note that K and $newsym(\hat{D})$ are disjoint.) A *canonical mapping* h is a function from $K \cup newsym(\hat{D})$ to itself that leaves the K symbols invariant (i.e., h is the identity function on K). A canonical mapping h is said to be *consistent* if $h(\hat{D})$ is consistent with all the views. □

Definition 6.3.3 (**Isomorphic Databases**) Let K be a set of symbols, and let D_1 and D_2 be two databases. We say that D_1 and D_2 are *K-isomorphic* (written as $D_1 \equiv_K D_2$) if there is a one-to-one mapping $\varphi : symbols(D_1) \to symbols(D_2)$ such that:

1. φ is the identity function on K,
2. φ maps $symbols(D_1) - K$ to $symbols(D_2) - K$, and
3. $\varphi(D_1) = D_2$.

□

We now state a lemma that relates the answers to the same query over two isomorphic databases.

Lemma 6.3.1 *Let Q be a conjunctive query, let K be a set of symbols that includes all the symbols used in Q, and let D_1 and D_2 be two databases. If $D_1 \equiv_K D_2$, then $Q(D_1) \equiv_K Q(D_2)$.* $\square$

Proof: Let Q be defined by rule H :- B. Assume $D_1 \equiv_K D_2$ and let φ be an associated isomorphism such that $\varphi(D_1) = D_2$. We would like to show $\varphi(Q(D_1)) = Q(D_2)$. Let t be a tuple in $Q(D_1)$. To show that $\varphi(t) \in Q(D_2)$, we need to find a substitution σ_2 such that $\sigma_2(B) \subseteq D_2$ and $\sigma_2(H) = \varphi(t)$. Since $t \in Q(D_1)$, there is a substitution σ_1 such that $\sigma_1(B) \subseteq D_1$ and $t = \sigma_1(H)$. Consider the substitution $\sigma_2 = \varphi \circ \sigma_1$. First, $\sigma_2(B) = \varphi(\sigma_1(B)) \subseteq \varphi(D_1) = D_2$. Second, $\sigma_2(H) = \varphi(\sigma_1(H)) = \varphi(t)$. Thus $\varphi(Q(D_1)) \subseteq Q(D_2)$. The converse can be shown analogously by using φ^{-1}. ∎

Since we will be dealing with databases that are consistent with the views, it is important to understand their relationship with interpretations of $\hat{D}$. The following lemma provides this relationship.

Lemma 6.3.2 *Given a set of views, let K be a set of symbols that includes the symbols used in the views and the conjunctive queries defining the views. Let D be a database consistent with all the views and let $\hat{D}$ be the canonical database. Then, D contains a database that is K-isomorphic with $h(\hat{D})$, for some canonical mapping h.* $\square$

Proof: Since D is consistent with all the views, for each view V (say defined by rule H :- B) and for each tuple $t \in V$, there is a substitution σ_t that turns B into tuples in D. Let D' be the collection of all such tuples (i.e., union of $\sigma_t(B)$ over all tuples t from each view). Similarly, by construction of $\hat{D}$, for each view tuple t, there is a substitution $\hat{\sigma}_t$ that turns B into tuples in $\hat{D}$. Clearly, σ_t and $\hat{\sigma}_t$ agree on the variables in B that also appear in H, and map them to view symbols. Also, $\hat{\sigma}_t$ maps the hidden variables to distinct symbols in $newsym(\hat{D}, t)$. Let $f_t : newsym(\hat{D}, t) \rightarrow symbols(D')$ be the function that composes the inverse of the restriction of $\hat{\sigma}_t$ over the hidden variables with σ_t. Let f be the function that is the identity function on K and that coincides with f_t over $newsym(\hat{D}, t)$ for each t. Clearly, $f(\hat{D}) = D'$. Next, we decompose f into two functions, h and φ, constructed as follows:

- For symbols $s \in newsym(\hat{D}$ such that $f(s) \in K$, then let $h(s) = f(s)$.
- For each symbol $s' \in symbols(D') - K$, consider the set $f^{-1}(s')$. Let $s_{s'}$ be a representative from that set. Then, for each $s \in f^{-1}(s')$, let $h(s) = s_{s'}$ and $\varphi(s_{s'}) = s'$.
- φ and h are the identity function on K.

It is easy to verify that:

- $f = \varphi \circ h$.
- h is a function from $K \cup newsym(\hat{D})$ to itself.
- φ is a one-to-one mapping: $symbols(h(\hat{D})) \rightarrow symbols(D')$

To summarize, we have constructed a database $D' \subseteq D$ and a canonical mapping h such that $D' \equiv_K h(\hat{D})$ (with isomorphism φ). ∎

We now return to the original question about the existence of consistent canonical mappings. The following theorem not only states that among all the possible interpretations of $\hat{D}$, there is always one that is consistent with all the views, but also makes the relationship between consistent databases and consistent interpretations of $\hat{D}$ explicit.

Theorem 6.3.1 *Given a set of views, let K be a set of symbols that includes the symbols used in the views and the conjunctive queries defining the views. Let $\hat{D}$ be the canonical database. Then, (1) any database D that is consistent with the views contains a database that is K-isomorphic to $h(\hat{D})$ for some consistent canonical mapping h, and (2) there is always a consistent canonical mapping h.* □

Proof: To show (1), let D be a database that is consistent with all the given views. Using Lemma 6.3.2, there must be some canonical mapping h and some subset $D' \subseteq D$ such that $h(\hat{D}) \equiv_K D'$. On the one hand, it follows from Lemma 6.3.1 that $Q_j(h(\hat{D})) \equiv_K Q_j(D')$ for every j, and since $Q_j(D')$ only uses symbols in K (because $\subseteq Q_i(D) = V_j$), it follows that $Q_j(h(\hat{D})) = Q_j(D') \subseteq Q_j(D) = V_j$. On the other hand, since $h(\hat{D})$ is one way to explain the presence of the tuples in all the views, it follows that $Q_j(h(\hat{D})) \supseteq V_j$ for every j. Therefore, $Q_j(h(\hat{D})) = V_j$ and $h(\hat{D})$ is consistent with all the views. To show (2), we use the assumption that an underlying database that is consistent with all the views exists. Then, (2) is a simple corollary of (1). ∎

To maintain a view (if it is self-maintainable), we can use the same idea as in Section 5.3 of finding a consistent database and using it to propagate an update to the view as if it were the actual database. A consistent canonical database is such a database, whose existence is guaranteed by Theorem 6.3.1. Thus, a solution to the maintenance question is to look for a consistent canonical mapping h and to propagate the given update to the view using $h(\hat{D})$ as the underlying database.

For the self-maintainability question, Theorem 5.4.2, the reduction theorem, must be extended to take into account the nonuniqueness of a canonical database that is consistent with the views. The following theorem formalizes the new reduction.

Theorem 6.3.2 *Let $V_1, \ldots, V_m$ be views, and for $i = 1, \ldots, m$, let Q_i be the conjunctive query that defines V_i. Let $\hat{D}$ be the canonical database and let U be a ground update to the underlying database. Let K be the set of symbols used in the V_i's, the Q_i's, and U. Let M be the set of consistent canonical mappings from $K \cup newsym(\hat{D})$ to itself. Then V_k is self-maintainable under U if and only if (1) For every $h \in M$, $Q_k(U(h(\hat{D})))$ contains no symbols from $newsym(\hat{D})$, (2) For every $h_1, h_2 \in M$, $Q_k(U(h_1(\hat{D}))) = Q_k(U(h_2(\hat{D})))$ holds, and (3) For every $h \in M$, $\bigwedge_i Q_i(D) = V_i$ implies $Q_k(U(D)) = Q_k(U(h(\hat{D})))$, for every database D that contains $h(\hat{D})$.* □

Proof:

IF: Let D_1 and D_2 be two databases that are consistent with the views. To show $Q_k(U(D_1)) = Q_k(U(D_2))$, we will elaborate on $Q_k(U(D_1))$, and $Q_k(U(D_2))$ follows similar reasoning. Since D_1 is consistent with the views, it follows from Theorem 6.3.1 that $D_1 \supseteq D'_1 \equiv_K h_1(\hat{D})$, for some D'_1 and some consistent h_1. Since it is always possible to construct a superset of $h_1(\hat{D})$ that is K-isomorphic to D_1, let $\hat{D}_1$ be such a superset. On the one hand, $\hat{D}_1$ is consistent with all the views since D_1 is, and it follows from (3) that $Q_k(U(\hat{D}_1)) = Q_k(U(h_1(\hat{D})))$. On the other hand, it is easy to see that $U(\hat{D}_1) \equiv_K U(D_1)$, and it follows from Lemma 6.3.1 that $Q_k(U(\hat{D}_1)) \equiv_K Q_k(U(D_1))$. Therefore $Q_k(U(D_1)) \equiv_K Q_k(U(h_1(\hat{D})))$, and it follows from (1) that $Q_k(U(D_1)) = Q_k(U(h_1(\hat{D})))$. Similarly for D_2, we can show that $Q_k(U(D_2)) = Q_k(U(h_2(\hat{D})))$ for some consistent h_2. Applying (2), we conclude that $Q_k(U(D_1)) = Q_k(U(D_2))$ and that V_k is self-maintainable under U.

ONLY-IF: Conversely, assume V_k is self-maintainable under U. To show *(1)*, assume there is a consistent h such that $Q_k(U(h(\hat{D})))$ contains some symbol s from $newsym(\hat{D})$. Consider a database D' obtained from $h(\hat{D})$ by replacing s with a new symbol s'. It is easy to see that not only D' is consistent with the views (since $h(\hat{D})$ is), but also $Q_k(U(D'))$ contains symbol s'. Obviously $Q_k(U(h(\hat{D})))$ cannot be identical to $Q_k(U(D'))$. Thus, there are two databases (namely $h(\hat{D})$ and D') that are both consistent with the views but that derive different instances of view V_k after the update, which contradicts the hypothesis that V_k is self-maintainable under U. Therefore *(1)* must holds. To show *(2)*, any pair of databases that are consistent with the views must derive the same view after update, in particular $h_1(\hat{D})$ and $h_2(\hat{D})$, where h_1 and h_2 are any consistent mappings. So *(2)* holds. To show *(3)*, let h be a consistent mapping and let D be a superset of $h(\hat{D})$ that is consistent with the views. Since V_k is self-maintainable under U, D and $h(\hat{D})$ must derive the same instance of view V_k after update. In other words, $Q_k(U(D)) = Q_k(U(h(\hat{D})))$. ∎

Note that condition *(3)* in Theorem 6.3.2 not only is decidable, but also can be decided, for each $h \in M$, in time polynomial in the size of the view instance and update instance. To see why, we use the same idea as in Theorem 5.4.2 of representing any superset of $h(\hat{D})$ by $D \cup h(\hat{D})$ (for an arbitrary D), and rewrite the implication as

$$Q_k(U(D \cup h(\hat{D}))) \not\subseteq Q_k(U(h(\hat{D}))) \Rightarrow \bigvee_i Q_i(D \cup h(\hat{D})) \not\subseteq V_i$$

Using the same technique based on query containment developed in Section 5.4, this implication can be solved in time polynomial in the size of the view instance and the update instance.

When we take functional dependencies into account in our problem, the results we obtain so far for the case without FD's need to be adjusted with the following modifications:

- For a canonical mapping h to be consistent, we require not only that $h(\hat{D})$ be consistent with all the views, but also that both $h(\hat{D})$ and $U(h(\hat{D}))$ satisfy the given FD's.

- Theorem 6.3.2 is extended by requiring the implication in condition *(3)* to hold for every database D that not only contains $h(\hat{D})$ but also satisfies the given FD's both before and after the update U is applied. This implication can be checked using the same technique as in Theorem 5.4.2.

To sum it up, for view maintenance, we can obtain an algorithm similar to Algorithm 5.3.1 except that the algorithm now uses a database $h(\hat{D})$ that is consistent with all the views (and with all the functional dependencies if any), instead of just $\hat{D}$. For deciding view self-maintainability, we can obtain an algorithm that computes all the consistent $h(\hat{D})$'s, compares the effect of the base update on the view when applied to the $h(\hat{D})$'s, and tests each of the $h(\hat{D})$'s using an algorithm similar to Algorithm 5.4.2.

While the use of projection in CQ views seems to make the problem considerably harder since the number of consistent canonical mappings h can be exponential in the worst case, results from Chapter 3 suggest that it does not have to be so. Chapter 3 showed that even with projection, self-maintainability of a single conjunctive-query view with no self-join can be efficiently decided with a simple query. Thus, an important future direction is to further refine our techniques and identify restrictions on the view that allow the problem to be solved more efficiently.

6.3.3 Unions of CQ Views

We now consider views that are unions of conjunctive queries which, for simplicity of discussion, do not have projections. We also ignore functional dependencies. For a view V defined by more than one rule (say by $mult(v)$ many rules), the presence of each tuple in the view can be explained by more than one set of facts ($mult(v)$ many sets to be exact). Each canonical database represents a particular way to explain all the facts in the views. Thus, given an instance of $V_1, \ldots, V_m$, the number d of canonical databases that can account for the contents of all the views is

$$mult(v_1)^{size(V_1)} \times \ldots \times mult(v_m)^{size(V_m)}.$$

Thus, like projections, unions introduce nonuniqueness of canonical databases. Canonical databases have the following properties:

- Any database that is consistent with all the views must contain some canonical database, which is necessarily consistent with all the views (since the queries defining the views are monotonic).
- Among all the canonical databases, there is at least one that is consistent with all the views (following the view realizability assumption).
- Not every canonical database is consistent with all the views.

The following example illustrates the nonuniqueness of canonical databases due to unions in views.

R	S	T
a, b	b, c	

D_1

R	S	T
a, b	b, c	a, b

D_2

R	S	T
a, b	a, b	b, c

D_3

R	S	T
	a, b	a, b
		b, c

D_4

Figure 6.3: The four canonical databases associated with the union views.

EXAMPLE 6.3.3 Consider views V_1 and V_2 defined by

$$v_1(X, Y, Z) \coloneq r(X, Y) \ \&\ s(Y, Z) \mid s(X, Y) \ \&\ t(Y, Z)$$
$$v_2(X, Y) \coloneq r(X, Y) \mid t(X, Y)$$

Consider the view instances $V_1 = \{(a, b, c)\}$ and $V_2 = \{(a, b)\}$. Figure 6.3 shows the four canonical databases which are obtained by considering all possible ways to choose a rule for each view. Among these canonical databases, only $\hat{D}_1$ and $\hat{D}_2$ are consistent with all the views. Both $\hat{D}_3$ and $\hat{D}_4$ generate (b, c), which is not in V_2. □

To maintain a view V_k, we can apply the same idea as in Section 5.3: propagate the update to V_k using some canonical database that is consistent with all the views. The choice of a canonical database is not important.

To answer the self-maintainability question, we must consider all the canonical databases that are consistent with all the views. Using these databases, the problem can be solved in a way that parallels the case of views with projections.

Theorem 6.3.3 *Let $V_1, \ldots, V_m$ be given views, and for $i = 1, \ldots, m$, let Q_i be the union of conjunctive queries that defines V_i. Let $\hat{D}_1, \ldots, \hat{D}_n$ be all the canonical databases that are consistent with all the views. Let U be an update to D, Then V_k is self-maintainable under U if and only if (1) $Q_k(U(\hat{D}_j)) = Q_k(U(\hat{D}_l))$ holds for every pair $\hat{D}_j$ and $\hat{D}_l$, and (2) $\bigwedge_i Q_i(D \cup \hat{D}_j) \subseteq V_i$ implies $Q_k(U(D \cup \hat{D}_j)) \subseteq Q_k(U(\hat{D}_j))$, for every database D and for every $\hat{D}_j$.* □

Proof:

IF: Let D_1 and D_2 be two databases that are consistent with the views. D_1 contains some $\hat{D}_j$ and D_2 contains some $\hat{D}_l$. Applying *(2)*, we infer that $Q_k(U(D_1)) = Q_k(U(\hat{D}_j))$ and $Q_k(U(D_2)) = Q_k(U(\hat{D}_l))$. Applying *(1)*, we conclude that $Q_k(U(D_1)) = Q_k(U(D_2))$. Thus, V_k is self-maintainable under U.

ONLY-IF: Conversely, assume V_k is self-maintainable under U. Any pair of databases that are consistent with the views must derive the same view after update, in particular

any pair $\hat{D}_j$ and $\hat{D}_l$. So *(1)* holds. To verify *(2)*, let D be a database and $\hat{D}_j$ a canonical database that is consistent with the views. Assume that $D \cup \hat{D}_j$ (call it D') is consistent with the views. Since V_k is self-maintainable under U, it follows that $Q_k(U(D')) = Q_k(U(\hat{D}_j))$.
∎

It it easy to see that Theorem 6.3.3 continues to hold if we consider only those canonical databases (consistent with the views) that are minimal. That is, we can ignore those canonical databases that contain some canonical databases consistent with the views.

In summary, to test self-maintainability of views that are unions of conjunctive queries:

- We compute all the minimal canonical databases that are consistent with the views.
- We then determine if updating any one of them has the same effect on all the views.
- For each minimal canonical database, we execute a query similar to what Algorithm 5.4.2 generates against the views and the canonical database.

6.4 Summary

In this chapter, we extended the general method developed in the previous chapter to cover a variety of situations in view self-maintenance. The required extensions were straightforward and the following results were obtained:

- When tuple updates are considered, and partial copies and subsets of the base relations are available, self-maintainability can still be tested in time polynomial in the size of the view instance, the update instance, any the base relation instance (if any). In particular, the tests are nonrecursive Datalog queries that can be generated at compile time.
- For views that are defined by conjunctive queries with projections or arithmetic comparisons, or unions of conjunctive queries, an obvious solution to the self-maintainability question is obtained and has a runtime complexity that is generally exponential in the size of the given relation instances.

We also applied the method to cases involving other data dependencies on the base relations. Testing view self-maintainability in the presence of multivalued dependencies reduces to testing containment of unions of conjunctive queries with negation (that applies to variable predicates), which can be solved with known algorithms ([LS93]). However, when the dependencies are embedded (e.g., inclusion dependencies), the resulting queries which we want to compare still involve negation but are no longer unions of conjunctive queries. Whether or not we can decide containment of these queries is still an open problem.

There remain many questions that deserve a closer look:

- Whether or when instance-specific containment of conjunctive queries involving arithmetic comparisons can be formulated as a query?

- Negation often appears only in the right hand side of the containment equation *DIFF* $\subseteq$ *INCON*. There are cases where all the rules involving negation can be eliminated without affecting the result. While containment of such queries is not generally known to be decidable or to have a polynomial solution, there may be special cases where the problem is more tractable. The following theorem is an example of such a special case.

  Theorem 6.4.1 *Consider a database D that includes relation S. Let P, Q, and R be queries over D. Let s be S's predicate. Assume the following:*

  – *P is independent of s or monotonic in s.*

  – *Both Q and R are independent of s or anti-monotonic in s.*

  Then $P \subseteq (Q \cup (R \text{ \& } \neg G))$ if and only if $P \subseteq Q$, where G is an atom with predicate s. □

  Proof: The *IF* part is trivial. For the *ONLY-IF* part, assume $P \not\subseteq Q$. There is a database D and a tuple t such that $t \in P(D)$ and $t \notin Q(D)$. Obtain D' by adding to D enough S-tuples so that $t \notin (R \text{ \& } \neg G)(D')$. But since P is monotonic in s, $P(D') \supseteq P(D)$ and thus, $t \in P(D')$. Also since Q is anti-monotonic in s, $Q(D') \subseteq Q(D)$ and thus, $t \notin Q(D')$. Therefore $P \not\subseteq (Q \cup (R \text{ \& } \neg G))$. ■

- When the view definitions involve projections or unions, the complexity of the results stems from the potentially large number of canonical databases and the cost to compute them. There may be situations where canonical databases come in small numbers or even unique, in which case a polynomial solution exists.

Chapter 7

Conclusion

7.1 Contribution of this Thesis

In this thesis, we proposed an approach to efficiently maintain a data warehouse by minimizing base access or even avoiding it totally. Such approach makes sense since data warehouses often have some degree of information redundancy that can be exploited in maintenance, and since accessing external data sources are usually much more expensive than accessing the warehouse.

The approach is based on the concept of *runtime view self-maintenance*, which consists of dynamically determining, at runtime, whether there is sufficient information to propagate a base update to a given view to maintain. While the approach is the most aggressive as it can be, efficiency of the solution is critical. We obtained the following results:

- We first considered maintenance of single CQ views with no self-joins, using no base relations (we called strict). The solutions are conjunctions of independent queries. They queries can be generated at compile time (Chapter 3).

- Next, we considered maintenance of single CQ views with no self-joins and no projections, in the presence of functional dependencies. The solutions are unions of conjunctive queries that can be generated at compile time (Chapter 4).

- We then considered maintenance of multiple CQ views with no projections under a wide variety of situations: under arbitrary updates, using functional dependencies, using partial base copies, and using base subsets. We developed a general method for this generalized view self-maintenance problem. The solutions are nonrecursive Datalog queries with negation that can be generated at compile time (Chapters 5 and 6).

- Finally, we considered maintenance of larger classes of views: views defined by conjunctive queries with arithmetic comparisons and projections, and unions of conjunctive queries. Our method provides a solution for the problem, although the solution is not in the form of a query that can be generated at compile time. The runtime complexity is generally exponential in the size of the relation instances (Chapter 6).

7.2 Future Work

In this thesis, we made a promising first step toward the practical use of runtime view self-maintenance in efficiently maintaining data warehouses. In order to fully realize the potential of this approach, there are many improvements and remaining issues we need to address. In the following, we outline five directions worth pursuing.

7.2.1 Identifying Efficient Special Cases

The results from Chapters 3 and 4 demonstrate that there are special cases of the view self-maintenance problem that admit simple and efficient solutions. More cases need to be identified and specialized methods to be developed. We are currently looking at views defined by conjunctive queries that are *acyclic* or have *fixed query width* (see [CR97]). This class of queries is interesting because it was shown in [CR97] that their containment can be decided in polynomial time, while the problem of decided conjunctive queries in general is known to be NP-complete ([CM77]). There is also another interesting special case we are studying for which we already obtained some partial results: maintaining CQ views with no self-joins where some of the base relations are materialized.

7.2.2 View Independent of Update

We mentioned at the beginning of this thesis that this problem can be viewed as special case of the view self-maintenance problem. As such, it appears to be a simpler problem. While it is definitely worth while to study this problem directly, it is also interesting to see whether the technical ideas developed in this thesis can be applied.

We also mentioned the close relationship between the view-independent-of-update problem and the problem of detecting constraint violations in distributed databases. It would be interesting to exchange results between these problems and to see how methods developed for one problem can be applied to the other problem.

7.2.3 Efficient solutions for query containment

Query optimization is a ubiquitous problem that underlies much work on information management. This thesis work is no exception. In particular, the queries that are generated using such a general technique as query containment tend to be very complex even if they are equivalent to much simpler queries. More powerful query optimization techniques than currently available need to be developed to simplify our solutions.

With the general method we developed in Chapter 5, we were often able to avoid the use of negation in the queries we would like to compare, when the view definitions do not involve negation. Yet, there are cases where negation cannot be totally avoided. The problem of deciding containment of queries with negation has unfortunately very limited results, and even in the cases where the problem is decidable, efficient solutions are little known. However, it may be possible to take advantage of the special forms of the queries we need to compare to come up with an efficient solution.

Finally, much work is needed to develop techniques for efficiently solving instance-specific query containment in general and for expressing it as efficient queries in particular.

7.2.4 Expressiveness of Views

In this thesis, we developed solutions to the view self-maintenance problem for different classes of views. These solutions may be efficient enough for their practical use, but it is also important to understand whether they can be as efficient as possible from a theoretical point of view. It is interesting not only to develop complexity lower bound for the view self-maintenance problem, but also to characterize classes of view definitions that do not admit a solution in the form of queries.

Data warehouses often make use of views that are defined with aggregates (e.g., averages), and perhaps with negation to a lesser extent. Maintaining such views poses new challenges and may require new methods to be developed.

7.2.5 Quantifying Self-Maintainability

The problem of view self-maintenance essentially involves three spaces of interest:

- S_1, the space of all possible situations,
- S_2, the space of all situations where a view can be unambiguously maintained, and
- S_3, the solution space, i.e., the space of those situations we can detect where a view is self-maintainable.

In this thesis, we emphasized completeness of the solution and required $S_3 = S_2$. However, there may be cases where it is worth trading off completeness for efficiency. It is therefore important to understand how close to S_2 S_3 can get, relative to S_1.

When a warehouse is designed, there are designs that are never self-maintainable (e.g., S_2 is empty) on the one extreme, and other designs that are always self-maintainable (e.g., $S_2 = S_1$) on the other extreme. In practice, we are probably concerned with choosing a design in between, and if the choice is driven by costs, it becomes important to understand how self-maintainable a given warehouse design is, that is, how close to S_1 is S_2.

Finally, we focused on solving the view self-maintenance problem when a given subset of the base relations is available but did not addressed the question of how to select such a subset. Choosing a good strategy for subset selection may involves evaluating the probability that a view is self-maintainable using a given subset, given that we have tried other subsets unsuccessfully.

Appendix A

Expressing Instance-Specific QC as a Query

In this appendix, we show that the containment $\mathit{DIFF} \subseteq \mathit{INCON}$ in Theorem 5.4.2 can be expressed as a (boolean) query in nonrecursive Datalog using the views and update relations as input.

Referring to Table 5.2, each of the queries *DIFF* and *INCON* can be expanded into a union of conjunctive queries, by eliminating the IDB predicates r'_j and r''_j. Note that the constant predicates $\hat{r}_j$ and $\hat{v}'_k$ do not have to be eliminated and can be treated as EDB predicates. The resulting containment equation has the form

$$\bigcup_i P_i \subseteq \bigcup_j Q_j \tag{A.1}$$

where each of the P_i's and Q_j's is a conjunctive query that uses both constant and variable EDB predicates, and where negation only applies to constant EDB predicates. Equation (A.1) can be reduced further to the conjunction (over i) of the following:

$$P_i \subseteq \bigcup_j Q_j \tag{A.2}$$

Thus, if for each i, we can express (A.2) as a safe query over the constant predicates, then the query *TEST* in Theorem 5.4.3 would be the conjunction over i of all such queries.

Unfortunately, since the individual conjunctive queries contain arithmetic comparisons, (A.2) is not always equivalent to the union (over j) of $P_i \subseteq Q_j$. To express (A.2) as a logical expression, we will first show how to express $P_i \subseteq Q_j$. Then, we will show that extending the result to the full union is not difficult, even though the extension involves more than just taking the disjunction.

To study the instance-specific query containment $P \subseteq Q$ where P and Q are conjunctive queries with $\neq$ comparisons and negated subgoals with constant predicates, we start with a theorem from [Gup94] (Theorem A.2.1 therein) that allows us to express containment of conjunctive queries with interpreted subgoals as a logical expression. We paraphrase it in the following, where *vars(X)* denotes the set of variables used in X:

Theorem A.0.1 *[Gup94] Consider the queries*

$$\begin{aligned} P: &\quad \textit{panic} \text{ :- } A \ \& \ B \\ Q: &\quad \textit{panic} \text{ :- } C \ \& \ D, \end{aligned}$$

where each of B and D represents a rectified conjunction of ordinary subgoals, and each of A and C represents a conjunction of interpreted subgoals such that $vars(A) \subseteq vars(B)$ and $vars(C) \subseteq vars(D)$. Let $\bar{U}$ denote $vars(B)$. Let M be the set of containment mappings: $D \mapsto B$. Then $P \subseteq Q$ if and only if

$$(\forall \bar{U}) \ [A \Rightarrow \bigvee_{h \in M} h(C)]. \tag{A.3}$$

□

Note the following about Theorem A.0.1:

- The condition for containment, given in Theorem A.2.1 from [Gup94] as $\bigvee_{h\in M}[A \Rightarrow h(C)]$, is wrong. For instance, if A is vacuously false, the containment should hold whether or not there is a containment mapping, but the condition from [Gup94] evaluates to false. In addition to this problem, there is another problem: for conjunctive queries with nontrivial heads, the condition given in Theorem A.2.1 from [Gup94] is sufficient but not necessary for the containment to hold. Consider the queries $P : r(0) \text{ :- } s(X) \ \& \ X = 0$ and $Q : r(Y) \text{ :- } s(Y)$ for instance. While P is obviously contained in Q, the theorem from [Gup94] predicts otherwise since there is no containment mapping from Q to P. However, for queries with trivial heads, the condition is both necessary and sufficient.

- The notion of rectification used in this theorem and in the remainder of this appendix is not the same as the one defined in Sections 2.4 and 4.2. Here, a conjunction (or a set) of subgoals is said to be *rectified* when no variables occur more than once among the subgoals, and no constant symbols are used. By contrast, in Sections 2.4 and 4.2, we only required that no variables occur more than once *within* each subgoal, and using the same variable in two different subgoals is allowed there.

- The predicate used is an interpreted subgoal, called an *interpreted* predicate, is allowed to be any boolean function that is computable, as long as the condition given in the theorem, a formula that uses interpreted predicates, is decidable. For instance, an interpreted subgoal may represent an arbitrary boolean combination of arithmetic comparisons over a dense domain.

- In our terminology, the predicates for the ordinary subgoals used in Theorem A.0.1 are variable predicates. Therefore, the theorem is not applicable when the queries contain subgoals with constant predicates.

Because of the latter limitation, we cannot apply Theorem A.0.1 to our instance-specific query containment problem. In the following, we will extend Theorem A.0.1 and develop analogous results for instance-specific containment. The rest of this appendix is divided into two parts. **Section A.1, *Expressing Instance-Specific Query Containment as a Logical Expression,*** shows how to reduce instance-specific query containment to a logical expression over the constant predicates. **Section A.2, *Making Certain Logical Expressions Safe,*** shows how to reduce a logical expression obtained in Section A.1 to an equivalent expression that is safe to evaluate.

A.1 Expressing Instance-Specific QC as a Logical Expression

In this section, we first show that $P \subseteq Q$, where each of P and Q is a conjunctive query (with $\neq$ and negation) that may use constant EDB predicates and where negation only applies to constant predicates, can be expressed as a logical expression over the constant predicates. We will provide logical expressions (see Table A.2) that are precise enough to allow us to analyze whether or not they are safe to evaluate and to express them as queries in the next section. We close this section by extending the results for Q's that are conjunctive queries to Q that are unions of conjunctive queries.

As mentioned before, the main limitation of Theorem A.0.1 is that it does not deal with subgoals with constant predicates that may be used in the queries. To remove this limitation, the idea is to treat subgoals (negated or not) with constant predicates as interpreted predicates. In fact, as mentioned in Section 2.5, a positive subgoal $g(\bar{X})$ with a constant predicate can be treated as the disjunction $\bigvee_{\bar{x}}(\bar{X} = \bar{x})$, and a negated subgoal $\neg g(\bar{X})$ with a constant predicate can be treated as the conjunction $\bigwedge_{\bar{x}}(\bar{X} \neq \bar{x})$, where $\bar{x}$ ranges over the tuples in the extension of g.

Also note the restriction in Theorem A.0.1 that the variables used in the interpreted subgoals must also appear among the ordinary subgoals. This restriction was needed to ensure that an interpreted subgoal uses only range-restricted variables. However, this restriction is not necessary if the interpreted subgoal represents a positive ordinary subgoal with a constant predicate.

We now state a lemma that extends Theorem A.0.1 by partially removing the "range-restricted variables" restriction for a class of interpreted subgoals (thus making it more general than Theorem A.0.1) and the requirement that ordinary subgoals be rectified. Note that while the latter extension is useful since it makes the containment condition simpler to analyze later on, the results in this appendix do not depend on it. Only the former extension is essential for the development that follows.

Lemma A.1.1 *Consider the queries*

$$\begin{aligned} P: &\quad \textit{panic} \text{ :- } A \;\&\; E \;\&\; B \\ Q: &\quad \textit{panic} \text{ :- } C \;\&\; F \;\&\; D, \end{aligned}$$

where

- *Each of B and D represents a conjunction of ordinary subgoals.*
- *D is rectified but B is not required to be so.*
- *Each of A and C represents a conjunction of interpreted subgoals.*
- *E (resp. F) represents a finite disjunction of zero or more equalities of the form $\bar{X}$ =<constant> (resp. $\bar{Y}$ =<constant>).*
- *$vars(A) \subseteq \bar{X} \cup vars(B)$ and $vars(C) \subseteq \bar{Y} \cup vars(D)$.*

Let $\bar{U}$ denote $vars(B)$, $\bar{V}$ denote $\bar{X} - vars(B)$, and $\bar{W}$ denote $\bar{Y} - vars(D)$. Let M be the set of containment mappings: $D \mapsto B$. Then $P \subseteq Q$ if and only if

$$(\forall \bar{U}, \bar{V})\ [A \wedge E \Rightarrow \bigvee_{h \in M} (\exists \bar{W})\ h(C \wedge F)].$$

$\square$

Proof: We first show the requirement in Theorem A.0.1 that B be rectified is not essential, and to this end, we refer to the proof of Theorem A.2.1 in [Gup94]. The *IF* part of the proof remains valid since it does not rely on the fact that B and D must be rectified. In the *ONLY-IF* part of the proof, we assume that (A.3) is false, that is, there is a substitution σ such that $\sigma(A)$ is true but $\sigma(h(C))$ is false for every $h \in M$. Consider the database $\mathcal{D}$ that consists of the tuples $\sigma(B)$. On the one hand, it is clear that when applied to $\mathcal{D}$, query P produces an answer. On the other hand, to see why query Q cannot produce any answer on $\mathcal{D}$, we assume it does. Then, there must be a substitution τ such that $\tau(C)$ is true and that each tuple in $\tau(D)$ is in $\mathcal{D}$. As a consequence of the latter fact, since D is rectified (that is, each variable in D only occur once and D has no constant symbols), there must be a containment mapping $h : D \mapsto B$ such that $\tau = \sigma \circ h$. Note that unlike the *ONLY-IF* proof in [Gup94], the existence of h only requires D to be rectified but not B, because there is no situation where we have to consider mapping a constant symbol in D or mapping a variable in D with multiple occurrences. Now since $\sigma(h(C))$ is false (by construction of σ), $\tau(C)$ is also false, which is a contradiction.

We now show the necessary and sufficient condition in the lemma. To this end, we use Theorem A.2.2 from [Gup94], an extension of Theorem A.0.1 to unions of conjunctive queries. We first write E as $\bigvee_i [\bar{V} = \bar{v}_i\ \&\ \bar{T} = \bar{t}_i]$ (where $\bar{T} = \bar{X} - \bar{V}$), and F as $\bigvee_j [\bar{W} = \bar{w}_j\ \&\ \bar{Z} = \bar{z}_j]$ (where $\bar{Z} = \bar{Y} - \bar{W}$). Let A_i (resp. C_j) be obtained from A (resp. C) after making the substitution $\bar{V} \rightarrow \bar{v}_i$ (resp. $\bar{W} \rightarrow \bar{w}_i$). Query P is equivalent to the union (over i) of the queries

$$P_i : panic \text{ :- } A_i\ \&\ (\bar{T} = \bar{t}_i)\ \&\ B$$

and query Q is equivalent to the union of the queries

$$Q_j : panic \text{ :- } C_j\ \&\ (\bar{Z} = \bar{z}_j)\ \&\ D$$

Applying Theorem A.2.1 from [Gup94], $\bigcup_i P_i$ is contained in $\bigcup_j Q_j$ if and only if

$$\bigwedge_i (\forall \bar{U})\ [A_i \wedge \bar{T} = \bar{t}_i \Rightarrow \bigvee_j \bigvee_{h:D \mapsto B} h(C_j \wedge \bar{Z} = \bar{z}_j)]$$

Since the right hand side of the implication does not depend on i and the containment mappings: $D \mapsto B$ do not depend on j, we rewrite the formula as follows, after pushing in $\bigwedge_i$ and $\bigvee_j$:

$$(\forall \bar{U})\ [\bigvee_i (A_i \wedge \bar{T} = \bar{t}_i) \Rightarrow \bigvee_{h:D \mapsto B} \bigvee_j h(C_j \wedge \bar{Z} = \bar{z}_j)]$$

Finally, by reintroducing $\bar{V}$ and W, we obtain:

$$(\forall \bar{U})\ [(\exists \bar{V})\ [A \wedge \bigvee_i (\bar{V} = \bar{v}_i \wedge \bar{T} = \bar{t}_i)] \Rightarrow \bigvee_{h:D \mapsto B} (\exists \bar{W})\ h(C \wedge \bigvee_j (\bar{W} = \bar{w}_j \wedge \bar{Z} = \bar{z}_j))]$$

■

Let us emphasize again that in Lemma A.1.1, although the interpreted subgoals E and F in the queries represent a disjunction of equalities, the containment condition uses E and F rather than their expanded form. We can now apply Lemma A.1.1 to solve the original containment $P \subseteq Q$, where P (resp. Q) is one of the conjunctive queries from *DIFF* (resp. *INCON*), by treating the set of positive subgoals with a constant predicate as subgoal E or F in Lemma A.1.1, and a negated subgoal with a constant predicate as an interpreted subgoal in A or C in Lemma A.1.1.

Referring to Table 5.2, query P takes the following generic form:

- *panic* :- $A(\bar{X})$ & $B(\bar{Y})$, where A represents a conjunction of subgoals with constant predicates (negated or not), and B a conjunction of non-negated subgoals with variable predicates. A uses variables $\bar{X}$ and B uses variables $\bar{Y}$, where $\bar{X}$ is a superset of $\bar{Y}$.

Query Q takes one of several possible forms, as shown in Table A.1.

The following example illustrates how to apply Lemma A.1.1 to express a containment as a logical expression.

EXAMPLE A.1.1 Consider the following queries, which are taken from above and Table A.1.

P : *panic* :- $A(\bar{X})$ & $B(\bar{Y})$
Q : *panic* :- $C(\bar{U},\bar{V})$ & $D(\bar{V},\bar{W})$ & $\neg S(\bar{U},\bar{V},\bar{W})$

where A represents a conjunction of subgoals with constant predicates (negated or not), C a non-negated subgoal with a constant predicate, $\neg S$ a negated subgoal with a constant predicate, and each of B and D a conjunction of non-negated subgoals with variable predicates. Also, $\bar{X}$ is a superset of $\bar{Y}$, and $\bar{U}$, $\bar{V}$, and $\bar{W}$ represent disjoint sets of variables. To apply Lemma A.1.1 for deciding $P \subseteq Q$, we need to rectify $D(\bar{V},\bar{W})$ into $D'(\bar{V},\bar{W},\bar{T})$ & $E(\bar{V},\bar{W},\bar{T})$, where $\bar{T}$ represents new variables, D' is rectified, and E equates

Form of Query Q	Explanation
$panic$:- $C(\bar{U},\bar{V})$ & $D(\bar{V},\bar{W})$ & $\neg S(\bar{U},\bar{V},\bar{W})$	C is a non-negated subgoal with some constant predicate, D is a conjunction of non-negated subgoals with variable predicates, and $\neg S$ is a negated subgoal with some constant predicate. This query form results from expanding rules (B'_i) from Table 5.2.
$panic$:- $D(U,\bar{V},\bar{W})$ & $D'(U',\bar{V}',\bar{W}')$ & $\bar{V}=\bar{V}'$ & $U \neq U'$	Each of D and D' is a non-negated subgoal with some variable predicate (in fact, they use the same predicate, but that is not important). This query form results from expanding rules $(L_{j\alpha\beta})$ from Table 5.2.
$panic$:- $C(U,\bar{V})$ & $\neg C'(U',\bar{V},\bar{W})$ & $D(U',\bar{V},\bar{W})$ & $U \neq U'$	C is a non-negated subgoal with some constant predicate, $\neg C'$ is a negated subgoal with some constant predicate, and D is a non-negated subgoal with some variable predicate. This query form results from expanding rules $(M_{j\alpha\beta})$ from Table 5.2.
$panic$:- $\neg C(U,\bar{V},\bar{W})$ & $D(U,\bar{V},\bar{W})$ & $\neg C'(U',\bar{V}',\bar{W}')$ & $D'(U',\bar{V}',\bar{W}')$ & $\bar{V}=\bar{V}'$ & $U \neq U'$	Each of $\neg C$ and $\neg C'$ is a negated subgoal with some constant predicate and each of D and D' is a non-negated subgoal with some variable predicate. This query form results from expanding rules $(M_{j\alpha\beta})$ from Table 5.2.

Table A.1: Possible forms of query Q.

a new variable with either a constant or some variable from $\bar{V}$ or $\bar{W}$. With all its ordinary subgoals (with variable predicates) rectified, query Q rewrites as follows:

$$Q: \quad panic \text{ :- } C(\bar{U},\bar{V}) \ \& \ D'(\bar{V},\bar{W},\bar{T}) \ \& \ \neg S(\bar{U},\bar{V},\bar{W}) \ \& \ E(\bar{V},\bar{W},\bar{T})$$

By treating $A(\bar{X})$ and $C(\bar{U},\bar{V})$ as disjunctions of equalities, and $\neg S(\bar{U},\bar{V},\bar{W})$ and $E(\bar{V},\bar{W},\bar{T})$ as interpreted subgoals, we can now apply Lemma A.1.1 to obtain the following condition for $P \subseteq Q$:

$$(\forall \bar{X}) \ [A(\bar{X}) \Rightarrow \bigvee_h (\exists \bar{U}) \ h(E(\bar{V},\bar{W},\bar{T}) \wedge S(\bar{U},\bar{V},\bar{W}) \wedge C(\bar{U},\bar{V}))]$$

where h ranges over containment mappings from $D'(\bar{V},\bar{W},\bar{T})$ to $B(\bar{Y})$. □

Form of subexpression F_Q	Explanation
$\bigwedge_h[\neg G_h(\bar{Y}) \vee$ $(\forall \bar{U})\ [C(\bar{U}, h(\bar{V})) \Rightarrow S(\bar{U}, h(\bar{V}), h(\bar{W}))]]$	h maps *rectify*(D) to B, and G_h is a conjunction of equalities that results from applying h to the equalities obtained from the rectification of D.
$\bigwedge_h[h(\bar{V}) = h(\bar{V}') \Rightarrow h(U) = h(U')]$	$h : D \ \&\ D' \mapsto B$.
$\bigwedge_h[C'(h(U'), h(\bar{V}), h(\bar{W})) \vee$ $(\forall U)\ [C(U, h(\bar{V})) \Rightarrow U = h(U')]]$	$h : D \mapsto B$.
$\bigwedge_h[C(h(U), h(\bar{V}), h(\bar{W})) \vee$ $C'(h(U'), h(\bar{V}'), h(\bar{W}')) \vee$ $h(\bar{V}) \neq (\bar{V}') \vee h(U) = h(U')]$	$h : D \ \&\ D' \mapsto B$.

Table A.2: Possible forms of logical subexpression F_Q.

The result of applying Lemma A.1.1 to express $P \subseteq Q$ as a logical expression in each of the cases above is summarized in the following theorem. Note that in order to apply Lemma A.1.1 correctly, the subgoals with variable predicates in Q must be rectified. Rectifying a set of subgoals simply involves introducing new variables and introducing additional subgoals that equate the new variables with constants or existing variables. The theorem is stated without proof.

Theorem A.1.1 *Let P (resp. Q) be one of the conjunctive queries from DIFF (resp. INCON). Using the characterization of P and Q above, the containment $P \subseteq Q$ can be expressed by the logical expression*

$$\neg(\exists \bar{X})[A(\bar{X}) \wedge F_Q]$$

where F_Q is one of the expressions shown in Table A.2, depending on the form Q takes. □

Finally, Lemma A.1.1 can be extended in the obvious way (much like the way Theorem A.2.2 extends Theorem A.2.1 in [Gup94]) to obtain a logical expression for the containment of a conjunctive query in a union of conjunctive queries. Based on this extension, which we do not show here, we can easily extend Theorem A.1.1 to the containment of P in *INCON*, as stated in the following theorem.

Theorem A.1.2 *Let P be one of the conjunctive queries from DIFF. The containment $P \subseteq INCON$ can be expressed by the logical expression*

$$\neg(\exists \bar{X})[A(\bar{X}) \wedge \bigwedge_Q F_Q] \tag{A.4}$$

where Q ranges over all the conjunctive queries in INCON and F_Q is a formula as specified in Theorem A.1.1. □

Form of Query A	Finite Query	Infinite Query
$A(\bar{Y}) : p(\bar{Y})$	$p(\bar{Y})$	
$A(\bar{Y}) : \neg p(\bar{Y})$		$\neg p(\bar{Y})$
$A(\bar{Y}) : (\forall \bar{U})\ p(\bar{U}, \bar{X}) \Rightarrow q(\bar{U}, \bar{Y})$	$(\exists \bar{U})\ q(\bar{U}, \bar{Y}) \wedge A(\bar{Y})$	$\neg(\exists \bar{U})\ p(\bar{U}, \bar{X})$
$A(\bar{Y}) : (\forall V)\ p(V, \bar{Y}) \Rightarrow V = \lambda$	$p(\lambda, \bar{Y}) \wedge A(\bar{Y})$	$\neg(\exists V)\ p(V, \bar{Y})$
$A(X_1, \bar{Y}) : (\forall V)\ p(V, \bar{Y}) \Rightarrow V = X_1$	$p(X_1, \bar{Y}) \wedge A(X_1, \bar{Y})$	$\neg(\exists V)\ p(V, \bar{Y})$
$A(\bar{Y}) : (\forall V)\ p(V, \bar{Y}) \Rightarrow V = X_2$	$p(X_2, \bar{Y}) \wedge A(\bar{Y})$	$\neg(\exists V)\ p(V, \bar{Y})$
$A(X_1, X_2) : X_1 \neq X_2$		$X_1 \neq X_2$
$A(X_1) : X_1 \neq \lambda$		$X_1 \neq \lambda$
Notation: p and q are safe queries, $\bar{Y}$ represents free variables, $\bar{X}$ is a subset of $\bar{Y}$, X_1 is a free variable not in $\bar{Y}$, X_2 is a free variable from $\bar{Y}$, and λ is a constant.		

Table A.3: Breaking up a query into a finite query and an infinite query.

A.2 Making Certain Logical Expressions Safe

In general, logical expressions such as (A.4) are not obviously safe. In fact, some conjuncts involve disjunction and negation. The queries that represent these conjunctions are not safe since they have an infinite number of answers.

In this section, we show that (A.4) can always be rewritten as an expression that is safe. This expression can be easily written as a safe, nonrecursive Datalog query (with negation and $\neq$ comparisons). Thus, the truth value of logical expression (A.4) can be determined in time polynomial in the size of the input.

A general transformation we often use is to rewrite $(\exists \bar{U}, \bar{V})[\alpha(\bar{U}) \vee \beta(\bar{V})]$ as the disjunction of the two formulas $(\exists \bar{U})\alpha(\bar{U})$ and $(\exists \bar{V})\beta(\bar{V})$. Thus we can eliminate the disjunctions in (A.4) to obtain a conjunction of formulas, each of which has the following form:

$$\neg(\exists \bar{Z})\ [A_1 \wedge A_2 \wedge \ldots \wedge A_n] \tag{A.5}$$

where each of the A's is a query in some variables from $\bar{Z}$ that may or may not be safe and that takes one of many forms. All the different forms of the A's are shown in the first column of Table A.3, where all free variables are drawn from $\bar{Z}$.

We now define the notion of finite and infinite queries we will use later:

- A *finite* query F in $\bar{X}$, denoted $F(\bar{X})$, is constructed from $p(\bar{X})$ where p is the predicate for a safe query, $(\exists \bar{Y})F'(\bar{X}, \bar{Y})$ where F' is a finite query, the conjunction of finite queries, or the conjunction of a finite query with any query in some subset of $\bar{X}$. Thus, a finite query always have a finite answer and can always be evaluated in finite time.
- An *infinite* query $I(\bar{X})$ is constructed from $\neg F(\bar{X})$ where F if a finite query, or from $\neq$ comparisons that involves variables in $\bar{X}$. Thus, there is always an infinite number of answers that satisfy an infinite query, and checking if a given value for $\bar{X}$ satisfies the query can be finitely evaluated.

Next, each of the A's is rewritten as a disjunction of a finite query and an infinite query. This rewriting is shown in Table A.3 for each form of the A's. After rewriting each of the A's from (A.5) and after eliminating the resulting $\vee$'s, we obtain a conjunction of formulas, each of which has the following form:

$$\neg(\exists\bar{X} \cup \bar{Y})[\bigwedge_i F_i(\bar{X}_i) \wedge \bigwedge_j I_j(\bar{Y}_j)] \tag{A.6}$$

where $\bar{X} = \bigcup_i \bar{X}_i$ and $\bar{Y} = \bigcup_j \bar{Y}_j$, the F_i's are finite queries, and the I_j's are infinite queries.

It is easy to verify that (A.6) is equivalent to the following safe formula:

$$\neg(\exists\bar{X}) \bigwedge_i F_i(\bar{X}_i) \wedge \bigwedge_k I_k(\bar{Y}_k)$$

where k ranges over those j such that $\bar{Y}_j \subseteq \bar{X}$.

Bibliography

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases,* Addison-Wesley, Reading, MA, 1995.

[ASU79a] A. V. Aho, Y. Sagiv, and J. D. Ullman. Equivalence of relational expressions. In *SIAM J. Computing* **8**:2, pp. 218–246, 1979.

[ASU79b] A. V. Aho, Y. Sagiv, and J. D. Ullman. Efficient optimization of a class of relational expressions. In *ACM Trans. on Database Systems* **4**:4, pp. 435–454, 1979.

[BC79] O. P. Buneman and G. K. Clemons. Efficiently monitoring relational databases. In *ACM Trans. on Database Systems* **4**:3, pp. 368–382, 1979.

[BCL89] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. In *ACM Trans. on Database Systems* **14**:3, pp. 369–400, 1989.

[BLT86] J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 61–71, Washington D. C., 1986.

[C*94] S. Chawathe, H. Garcia-Molina, J. Hammer, Y. Papakonstantinou, J. D. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. IPSJ Conf.,* pp. 7–18, Tokyo, Oct. 1994.

[CM77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relation databases. In *Proc. 9th Annual ACM Symposium on the Theory of Computing,* pp. 77–90, 1977.

[CR97] A. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *Proc. 6th Int. Conf. on Database Theory,* pp. 56–70, Delphi, Greece, 1997.

[CV92] S. Chaudhuri and M. Y. Vardi. On the equivalence of Datalog programs. In *Proc. 11th ACM Symp. on Principles of Database Systems*, pp. 55–66, 1992.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. 17th Int. Conf. on Very Large Data Bases,* pp. 577–589, 1991.

[DS92] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. In *Technical Report TRCS 92-18,* University of California, Santa Barbara, 1992.

[DT92] G. Dong and R. Topor. Incremental evaluation of datalog queries. In *Proc. 4th Int. Conf. on Database Theory,* pp. 282–296, Berlin, Germany, 1992.

[Elk90] C. Elkan. Independence of logic database queries and updates. In *Proc. 9th ACM Symp. on Principles of Database Systems*, pp. 154–160, 1990.

[G*94] A. Gupta, Y. Sagiv, J. D. Ullman, and J. Widom. Constraint checking with partial information. In *Proc. 13th ACM Symp. on Principles of Database Systems*, pp. 45–55, 1994.

[GB95] A. Gupta and J. A. Blakeley. Using partial information to update materialized views. In *Information Systems* **20**:8, pp. 641–662, 1995.

[GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *EDBT,* pp. 140–144, Avignon, France, 1996.

[GKM92] A. Gupta, D. Katiyar, and I. S. Mumick. Counting solutions to the view maintenance problem. In *Proc. JICSLP Workshop on Deductive Databases,* pp. 185–194, 1992.

[GLT97] T. Griffin, L. Libkin, and H. Trickey. An improved algorithm for the incremental recomputation of active relational expressions. In *IEEE Trans. on Knowledge and Data Engineering* **9**:3, pp. 508–511, 1997.

[GM*95] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. In *2nd Workshop on Next-Generation Information Technologies and Systems,* Naharia, Israel, 1995.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. In *IEEE Data Engineering Bulletin, Special Issue on Materialized Views & Data Warehousing* **18**:2, June 1995.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 157–166, Washington D. C., June 1993.

[GU92] A. Gupta and J. D. Ullman. Generalizing conjunctive query containment for view maintenance and integrity constraint verification. In *Proc. JICSLP Workshop on Deductive Databases,* pp. 195, 1992.

[Gup94] A. Gupta. *Partial Information Based Integrity Constraint Checking,* Stanford University Technical Report CS-TR-95-1534, Ph.D. Thesis, Stanford, Nov. 1994.

[H*95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford data warehousing project. In *IEEE Data Engineering Bulletin* **18**:2, pp. 41–48, June 1995.

[HD92] J. V. Harrison and S. Dietrich. Maintenance of materialized views in a deductive database: an update propagation approach. In *Proc. JICSLP Workshop on Deductive Databases,* pp. 56–65, 1992.

[Huy96a] N. Huyn. Efficient view self-maintenance. In *Proc. Int. Workshop on Materialized Views: Techniques and Applications,* pp. 17–25, Montreal, Quebec, 1996.

[Huy96b] N. Huyn. Efficient self-maintenance of materialized views. Unpublished Technical Report, available as `http://www-db.stanford.edu/pub/papers/vsm-2-tr.ps`, 1996.

[Huy96c] N. Huyn. Exploiting dependencies to enhance view self-maintainability. Unpublished Technical Report, available as `http://www-db.stanford.edu/pub/papers/fdvsm.ps`, 1996.

[Huy97a] N. Huyn. Efficient complete local tests for conjunctive-query constraints with negation. In *Proc. 6th Int. Conf. on Database Theory,* pp. 83–97, Delphi, Greece, 1997.

[Huy97b] N. Huyn. Multiple-view self-maintenance in data warehousing environments. In *Proc. 23rd Int. Conf. on Very Large Data Bases,* pp. 26–35, Athens, Greece, 1997.

[Huy97c] N. Huyn. Maintaining global integrity constraints in distributed databases. In *Constraints Journal, Special Issue on Constraints and Databases,* Kluwer Academic Publishers, 1998.

[IK93] W. H. Inmon and C. Kelley. *Rdb/VMS: Developing the data warehouse,* QED Publishing Group, Boston, Massachusetts, 1993.

[JK83] D. S. Johnson and A. Klug. Optimizing conjunctive queries that contain untyped variables. In *SIAM J. Computing* **12**:4, pp. 616–640, 1983.

[JK84] D. S. Johnson and A. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. In *J. Computer and System Sciences* **28**:1, pp. 167–189, 1984.

[JMS95] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues in the chronicle data model. In *Proc. 14th ACM Symp. on Principles of Database Systems*, pp. 113–124, 1995.

[Klu88] A. Klug. On conjunctive queries containing inequalities. In *J. ACM* **35**:1, pp. 146–160, 1988.

[Kuc91] V. Kuechenhoff. On the efficient computation of the difference between consecutive database states. In *Proc. Int. Conf. on Deductive and Object-Oriented Databases,* pp. 478–502, 1991.

[LS93] A. Levy and Y. Sagiv. Queries independent of updates. In *Proc. 19th Int. Conf. on Very Large Data Bases,* pp. 171–181, Dublin, Ireland, 1993.

[Mey92] R. van der Meyden. The complexity of querying indefinite data about linearly ordered domains. In *Proc. 11th ACM Symp. on Principles of Database Systems*, pp. 331–345, 1992.

[Pai84] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In Gallaire H., Minker J. and Nicolas J. M., editors, *Advances in Data Base Theory, vol. 2,* pp. 171–209, Plenum Press, New York, 1984.

[Pap96] Y. Papakonstantinou. *Query Processing in Heterogeneous Information Sources.* Ph.D. Thesis, Computer Science Department, Stanford University, 1996.

[Q*96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. 4th Int. Conf. on Parallel and Distributed Information Systems,* Miami Beach, FL, Dec. 1996.

[Qua97] D. Quass. *Materialized Views in Data Warehouses.* Ph.D. Thesis, Computer Science Department, Stanford University, 1997.

[QW91] X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. In *IEEE Trans. on Knowledge and Data Engineering* **3**:3, pp. 337–341, 1991.

[RED] Red Brick Systems. *Red Brick Warehouse,* 1995.

[RSUV89] R. Ramakrishnan, Y. Sagiv, J. D. Ullman, and M. Y. Vardi. Proof-tree transformations and their applications. In *Proc. 8th ACM Symp. on Principles of Database Systems*, pp. 172–182, 1989.

[RSUV93] R. Ramakrishnan, Y. Sagiv, J. D. Ullman, and M. Y. Vardi. Logical query optimization by proof-tree transformation. In *J. Computer and System Sciences* **47**:1, pp. 222–248, 1993.

[Sag87] Y. Sagiv. Optimizing datalog programs. In *Proc. 6th ACM Symp. on Principles of Database Systems*, pp. 349–362, 1987.

[SI84] O. Shmueli and A. Itai. Maintenance of views. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 240–255, 1984.

[SJ96] M. Staudt and M. Jarke. Incremental maintenance of externally materialized views. In *Proc. 22nd Int. Conf. on Very Large Data Bases,* pp. 75–86, Mumbai, India, 1996.

[Sto75] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 65–78, San Jose, CA, 1975.

[SY80] Y. Sagiv and M. Yannakakis. Equivalences among expressions with the union and difference operators. In *J. ACM* **27**:4, pp. 633–655, 1980.

[TB88] F. W. Tompa and J. A. Blakeley. Maintaining materialized views without accessing base data. In *Information Systems* **13**:4, pp. 393–406, 1988.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes 1 and 2,* Computer Science Press, Rockville, MD, 1989.

[UO92] T. Urpi and A. Olive. A method for change computation in deductive databases. In *Proc. 18th Int. Conf. on Very Large Data Bases,* pp. 225–237, 1992.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing set-oriented production rules as an extension to Starburst. In *Proc. 7th Int. Conf. on Very Large Data Bases,* pp. 275–285, Barcelona, Spain, 1991.

[Z*95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 316–327, San Jose, CA, 1995.

[Zan86] C. Zaniolo. Safety and compilation of nonrecursive Horn clauses. In *Proc. 1st Int. Conf. Expert Database Systems,* pp. 167–178, Benjamin-Cummings, Menlo Park, CA, 1986.

[ZWG97] Y. Zhuge, J. Wiener, and H. Garcia-Molina. Multiple view consistency for data warehousing. In *Proc. 13th Int. Conf. on Data Engineering,* pp. 289–300, Birmingham, UK, 1997.