7-8-2009

Inconsistency and Incompleteness in Relational Databases and Logic Programs

Navin Viswanath

INCONSISTENCY AND INCOMPLETENESS IN RELATIONAL DATABASES AND LOGIC PROGRAMS

by

NAVIN VISWANATH

Under the direction of Rajshekhar Sunderraman

ABSTRACT

The aim of this thesis is to study the role played by negation in databases and to develop data models that can handle inconsistent and incomplete information. We develop models that also allow incompleteness through disjunctive information under both the CWA and the OWA in relational databases. In the area of logic programming, extended logic programs allow explicit representation of negative information. As a result, a number of extended logic programs have an inconsistent semantics. We present a translation of extended logic programs to normal logic programs that is more tolerant to inconsistencies. Extended logic programs have also been used widely in order to compute the repairs of an inconsistent database. We present some preliminary ideas on how source information can be incorporated into the repair program in order to produce a subset of the set of all repairs based on a preference for certain sources over others.

INDEX WORDS: Incompleteness, Inconsistency, Logic programming, Negation, Nonmonotonic reasoning

INCONSISTENCY AND INCOMPLETENESS IN RELATIONAL DATABASES AND LOGIC PROGRAMS

by

NAVIN VISWANATH

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2009

INCONSISTENCY AND INCOMPLETENESS IN RELATIONAL DATABASES AND LOGIC PROGRAMS

by

NAVIN VISWANATH

Committee Chair: Rajshekhar Sunderraman

Committee: Yanqing Zhang
Anu Bourgeois
Yichuan Zhao

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
August 2009

ACKNOWLEDGMENTS

I would like to thank my parents for being a constant source of encouragement during my days in graduate school. Thank you for being patient even though you feared I might remain a student for the rest of my life; those days are now over, finally!

I am indebted to my advisor Dr. Rajshekhar Sunderraman for teaching me everything I know about databases and logic programming. His papers have been entirely responsible for inspiring me to pursue a Ph.D. I hope that his work will continue to inspire many other students like me. He has been my friend, philosopher and guide through many good and bad days. I will fondly remember the innumerable conversations in his office. Through these conversations he taught me not only how to do research but also lessons for life that have left an indelible mark on me. I would also like to thank my committee members Dr. Yanqing Zhang, Dr. Anu Bourgeois, and Dr. Yichuan Zhao for patiently listening to me and assessing my work.

I would like to thank God Almighty, for showing me this path and being kind to me at every step.

There are a number of friends who have at some point or the other been a part of this process and influenced me in their own little way. There are a few I cannot fail to mention: my friend Akshaye Dhawan, who has been with me through the ups and downs. Thank you for those wonderful lunch and coffee conversations over six long years. I cannot think of anyone better to talk to when the ranting got too technical! Thank you for listening and keeping me motivated. I would like to thank my friends Ajay Tripathi and Mayur Chhabra for reminding me that there is a life outside school. Thank you for keeping me sane! I would like to thank Sira Rao who by way of example taught me about the kind of hard work, determination and single-minded focus that was needed to get here.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1.

INTRODUCTION

Typically, relational databases operate under the *Closed World Assumption* (CWA) of Reiter [64]. The CWA is a meta-rule that says that given a knowledge base KB and a sentence P, if P is not a logical consequence of KB, assume $\sim P$ (the negation of P). Thus, we explicitly represent only positive facts in a knowledge base. A negative fact is implicit if its positive counterpart is not present. Under the CWA we presume that our knowledge about the world is complete i.e. there are no "gaps" in our knowledge of the real world. *The Open World Assumption* (OWA) is the opposite point of view. Here, we "admit" that our knowledge of the real world is incomplete. Thus we store everything we know about the world - positive and negative. Consider a database which simply contains the information "Tweety is a bird". Assume that we want to ask this database the query "Does Tweety fly?". Under the CWA, since we assume that there are no gaps in our knowledge, every query returns a yes/no answer. In this case we get the answer "no" because there is no information in the database stating that Tweety can fly. However, under the OWA, the answer to the query is "unknown". i.e. the database does not know whether Tweety flies or not. We would obtain a "no" answer to this query under the OWA only if it was explicitly stated in the database that Tweety does not fly.

Current implementations of relational databases adopt the CWA; and for good reason. The negative facts generally turn out to be much larger than the positive facts and it may be unfeasible to store all of it in the database. A typical example is an airline database that records the flights between cities. If there is no entry in the database of a flight between city X and city Y, then it is reasonable to conclude that there is no flight between the cities. Thus for many application domains the Closed World Assumption is appropriate. However, there are a number of domains where the CWA is not appropriate. A prime example is databases that require domain knowledge. For example, consider a biological database that

stores pairs of neurons that are connected to each other. If we were to ask this database the query "Is neuron N1 connected to neuron N2?" and this information was not available in the database, is "no" an appropriate answer? What if we do not know yet whether N1 is connected to N2? Then surely the answer "no" is misleading. A more appropriate answer would be "unknown" which we would obtain under the OWA.

A problem that has recently been seeing a resurgence of interest in the database community is that of incomplete information. In an era where it is common to see data integrated from different sources, the importance of this problem cannot be overemphasized. The problem of incorporating incomplete information in a logical data model such as the relational model was first emphasized by Codd. The earliest efforts in this area were targeted at incorporating a "null value" in the database. This null could be interpreted in a number of different ways. Perhaps the most important work that laid the foundations for the problem of query processing in a database containing incomplete information is that of Imielinski and Lipski [41]. This work introduced a data model, called the *c-tables*, that extended the relational model of Codd and could represent any form of incompleteness. They also proposed an algebra for query processing and laid out the definitions for what constitutes "correct" query answers in the presence of incomplete information.

Deductive databases and logic programming have widely been recognized as expressive knowledge representation formalisms. The idea of using first order predicate logic as a programming language was first introduced by van Emden and Kowalski in [79]. In this paper they provide a semantics for class of logic programs called the *Horn programs*. A number of extensions were found to be necessary in order to gain expressivity. Initially, the Horn logic programs were extended to include negation in the body of rules. The semantics of such programs have been an active area of research for a number of years and two semantics that have emerged as being the most widely accepted by the research community are the well-founded semantics [81] and the stable model semantics [32]. Gelfond and Lifschitz showed in [33] that even greater expressivity can be obtained by including a different kind of negation

in logic programs. Logic programs containing two types of negation are called *extended logic programs*. The two kinds of negation in extended logic programs are *explicit negation* and *default negation*. Default negation is the ordinary negation considered in semantics such the well-founded semantics and the stable model semantics. The semantics for such programs were given by the *answer set semantics*. This led to the formulation of a number of different semantics for extended logic programs [11, 73, 42, 5].

The relational model is devoid of semantics. One way of incorporating semantics into the model is by introducing constraints on the schema. Typically a constraint is a statement in first order logic to be satisfied by any instance of the database. For instance, the statement that every individual has a unique social security number is one such constraint. However, it is very often the case that these constraints are violated. In such a scenario, it is not clear how query answering should be accomplished on such a database. This problem is particularly prevalent in the data integration scenario. When data is integrated from a number of sources it may happen that while the data from each source is independently consistent, inconsistencies may creep into the integrated database. However, even in such inconsistent databases there is typically a large amount of useful information and the inconsistency is usually limited to a small portion of the database. In such a situation we may want to retrieve answers to queries only from the consistent portion of the database. This is known as the problem of *consistent query answering*(CQA). Broadly, there are two ways to achieve this goal. One of them is *query rewriting*, in which the query is rewritten so that it retrieves only consistent answers. Another approach to the problem is to *repair* the database. A repair of an inconsistent database is obtained by performing "minimal updates" to restore consistency. As can be seen, an inconsistent database may have an exponential number of repairs. A consistent query answer is an answer that is true in every repair of the database.

My research has touched upon all of the above mentioned problems. A significant portion of my research has focused on the problems of incompleteness and inconsistency under

the OWA. We develop data models based on the OWA that can handle inconsistent and incomplete information.

The outline of this dissertation is as follows. The dissertation is divided into four parts. Part I presents a background to the dissertation. Chapter 2 is an introduction to logic programming. Chapter 3 briefly introduces constraints in relational databases. Chapter 4 introduces a data model based on the OWA, called the *paraconsistent data model*. Part II of the dissertation concerns incompleteness. Chapter 5 introduces a data model based on the CWA that handles incomplete information. Chapter 6 presents another data model based on the OWA that handles incomplete information. It may be noted that both these models are complete i.e. for any set of possible worlds there is an instance in these representation formalisms. Part III of the dissertation studies *nonmontonic reasoning*. Chapter 7 introduces a data model called *d-relations* that operates under the OWA and has two forms of negation. One is the explicit negation found in the paraconsistent data model and the other is a nonmonotonic form of negation. Chapter 8 studies that problem of inconsistency in extended logic programs. We introduce a technique of translating extended logic programs to normal logic programs so that inconsistencies may be avoided. Part IV of the dissertation concerns the problem of consistent query answering in inconsistent databases. Chapter 9 is a treatment of functional dependencies under the OWA. We present a data model that handles constraints in the open world setting. In chapter 10, we present a method by which lineage information can be incorporated into the extended logic program that computes the repairs of the inconsistent database. As a result, the number of repairs produced is reduced largely.

Part I

Background

CHAPTER 2.

LOGIC PROGRAMMING

This chapter introduces the field of logic programming. Logic programming has emerged as a very expressive tool for knowledge representation. We introduce the basic concepts of logic programming and focus exclusively on the declarative semantics of logic programs. The reader is referred to [50] for a more detailed description of the operational semantics of logic programs.

2.1 Definite Logic Programs

We first introduce *definite logic programs*, which are logic programs that do not contain negation. The semantics of definite logic programs is given by the T_P operator of van Emden and Kowalski in [79]. A definite logic program is a set of *Horn* clauses. We first introduce some of the basic structures before defining a Horn clause.

A *term* is a constant, a variable or a complex term of the form $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and f is a *function symbol* with finite arity $n \geq 0$. An *atom* is a formula of the language of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of finite arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms. A *literal* is either an atom or its negation, denoted by $p(t_1, \ldots, t_n)$.

A definite logic program is a set of rules of the form

$$A \leftarrow B_1, \ldots, B_n \tag{2.1}$$

where $A, B_1, \ldots, B_n$ are atoms. Here A is called the head of the rule and the conjunction $B_1 \wedge \ldots \wedge B_n$ is called the body of the rule. Given a logic program P, the *Herbrand universe* of P, denoted U_P, is the set of all possible ground terms constructed recursively using the constants and function symbols occuring in P. The *Herbrand base* of P, denoted HB_P, is the set of all possible ground atoms whose predicate symbols occur in P and whose arguments

are elements of U_P. A term, atom, literal, rule or program is ground if it is free of variables. A *ground instance* of a rule is obtained by replacing the variables in a program with elements from U_P in every possible way. A ground program is the union of the ground instances of the rules in the program.

2.1.1 Model-theoretic semantics

A Herbrand interpretation I of P is any subset of the Herbrand base of P. A Herbrand interpretation simultaneously associates, with every n-ary predicate symbol in P, a unique n-ary relation over U_P.

1. A ground atomic formula A is true in a Herbrand interpretation I iff $A \in I$.
2. A ground negative literal $\neg A$ is true in I iff $A \notin I$.
3. A ground clause $L_1 \vee \ldots \vee L_m$ is true in I iff at least one literal L, is true in I.
4. In general a clause C is true in I iff every ground instance $C\sigma$ of C is true in I. ($C\sigma$ is obtained by replacing every occurrence of a variable in C by a term in U_P. Different occurrences of the same variable are replaced by the same term.)
5. A set of clauses $\mathbf{A}$ is true in I iff each clause in $\mathbf{A}$ is true in I.

A literal, clause, or set of clauses is false in I iff it is not true. If $\mathbf{A}$ is true in I, then we say that I is a Herbrand model of $\mathbf{A}$. Let $\mathbf{M}(\mathbf{A})$ be the set of all Herbrand models of A; then $\cap\mathbf{M}(\mathbf{A})$, the intersection of all Herbrand models of A, is itself a Herbrand interpretation of A. This holds for any set of clauses A even if A is inconsistent. If A is a consistent set of Horn clauses then $\cap\mathbf{M}(\mathbf{A})$ is itself a Herbrand model of A. More generally, Horn clauses have the *model intersection property*: If $\mathbf{L}$ is any nonempty set of Herbrand models of A then $\cap\mathbf{L}$ is also a model of A, and is the least such model of A.

$\{P(a) \vee P(b)\}$, where a and b are constants, is an example of a non-Horn sentence which does not have the model-intersection property: $\{\{P(a)\}, \{P(b)\}\}$ is a nonempty set of models, yet its intersection $\emptyset$ is a Herbrand interpretation which is not a model.

2.1.2 Fixpoint semantics

The fixpoint semantics of a definite logic program P is given by means of an operator T_P on interpretations. The *least fixpoint* of T_P is the *least model* of P. This result relies on the fact that the operator T_P is monotonic and hence possesses a least fixpoint. T_P is monotonic since for any interpretations I_1 and I_2, such that $I_1 \subseteq I_2$, $T(I_1) \subseteq T(I_2)$. The least fixpoint is given by

$$\cap\{I : T_P(I) \subseteq I\} \tag{2.2}$$

The definition of T_P is given as follows:
$T(I)$ contains a ground atomic formula $A \in H_P$ iff for some ground instance $C\sigma$ of a clause $C \in P$, $C\sigma = A \leftarrow B_1, \ldots B_n$ and $B_1, \ldots B_n \in I$, $n \geq 0$.

For a definite logic program P, let $\mathbf{M(P)}$ be its Herbrand models and let $\cap\mathbf{M(P)}$ be its least model. Let $\mathbf{C(P)}$ be the set of all interpretations closed under T_P, i.e., $I \in \mathbf{C(P)}$ iff $T_P(I) \subseteq I$. We need to show that $\cap\mathbf{M(P)} = \cap\mathbf{C(P)}$. It is easier to show that $\mathbf{C(P)} = \mathbf{M(P)}$.

Theorem 2.1.1. *If P is a definite logic program then $\mathbf{M(P)} = \mathbf{C(P)}$, i.e. $\models_I P$ iff $T(I) \subseteq I$, for all Herbrand interpretations I of P.*

Proof. ($\models_I P$ implies $T_P(I) \subseteq I$). Suppose I is a model of P. We want to show that if $A \in T_P(I)$ then $A \in I$. Assume that $A \in T_P(I)$. Then by the definition of T_P, there is a clause $C \in P$ such that $C\sigma = A \leftarrow B_1, \ldots B_n$ and $B_1, \ldots B_n \in I$. Since I is a model of P, $C\sigma$ is true in I which means that A is true in I since $\neg B_1, \ldots \neg B_n$ are false in I. Therefore $A \in I$.
($T_P(I) \subseteq I$ implies $\models_I P$). Suppose that I is not a model of P. Then for some clause $C \in P$, $C\sigma = A \leftarrow B_1, \ldots B_n$ is false in I, i.e., $B_1, \ldots B_n \in I$ and $A \notin I$. But by the definition of T_P, since $B_1, \ldots B_n \in I$, $A \in T_P(I)$. Thus $T_P(I) \not\subseteq I$. □

It can also be shown that the least model is the limit of the increasing, possibly infinite sequence of iterations $\emptyset, T_P(\emptyset), T_P(T_P(\emptyset)) \ldots$.

There is a standard notation used to denote elements of the sequence of interpretations constructed for P. Namely:

$$\begin{aligned} T_P \uparrow 0 &= \emptyset \\ T_P \uparrow i+1 &= T_P(T_P \uparrow i) \\ T_P \uparrow \omega &= \bigcup_{i=0}^{\infty} T_p \uparrow i. \end{aligned}$$

We show the iterations of the T_P operator with an example.

Example 2.1.1

Consider the definite logic program

$odd(s(0))$.

$odd(s(s(X))) \leftarrow odd(X)$.

$$\begin{aligned} T_P \uparrow 0 &= \emptyset \\ T_P \uparrow 1 &= \{odd(s(0))\} \\ &\vdots \\ T_P \uparrow \omega &= \{odd(s^n(0)) \mid n \in \{1,3,5,\ldots\}\} \end{aligned}$$

2.2 Introducing Negation

This section describes some of the results in extending Horn clause programs to include negation in the body of clauses. We call such logic programs *general logic programs* or *normal logic programs*. Work in this area has proceeded in two directions : one is the *program completion* approach and the other is the search for a *canonical model* of the logic program. A general logic program is a set of clauses of the form

$$A \leftarrow B_1, \ldots B_n, \sim C_1, \ldots, \sim C_m \tag{2.3}$$

Here the symbol $\sim$ denotes negation. This is different from a different form of negation $\neg$ that we will introduce later.

One of the major issues with introducing negation in the body of the program clauses is that there is no longer a unique least model of the program. Instead there are several *minimal models*. We illustrate this with an example. This is an example from [77]. The program describes two bus lines, red and blue, each of which runs between pairs of cities. Thus $blue(X, Y)$ means there is a bus of the blue line between cities X and Y. The president of the red line wants to find if there is a pair of cities such that there is a bus of the red line between them, but there is no way to go from one city to the other on the blue line even through a sequence of intermediate stops.

Example 2.2.1

$$
\begin{aligned}
red(1,2) &\leftarrow \\
blue(1,2) &\leftarrow \\
red(2,3) &\leftarrow \\
bluePath(X,Y) &\leftarrow blue(X,Y). \\
bluePath(X,Y) &\leftarrow blue(X,Z), bluePath(Z,Y). \\
monopoly(X,Y) &\leftarrow red(X,Y), \sim bluePath(X,Y).
\end{aligned}
$$

This program has two minimal models. They are $\{bluePath(1,2), monopoly(2,3)\}$ and $\{bluePath(1,2), bluePath(2,3), bluePath(1,3)\}$ along with the facts. The question now is to decide which of these two models should be accepted as the intended model of the program. It is easy to see that the first model appears to be the intended one. The only blue path is the one that follows from the data and using the first rule for $bluePath$. Then the monopoly fact follows from the last rule. The second model seems to materialize $bluePath(2,3)$ and $bluePath(1,3)$ from nowhere.

2.2.1 Program completion

We first describe the program completion approach. Much of the results here are based on the work of Clark [20]. The basic idea in program completion is to treat the implication in the rule as only the "if" half of a first order theory. We obtain a completion by converting the "if" to an "if and only if" along with a set of equality axioms and unique names assumptions. The classical, 2-valued logical consequences of this theory should be the logical conclusions of the program. The program completion approach is very attractive. First, it is logically correct. Secondly, it lends itself very elegantly to a computation mechanism known as SLDNF resolution and thus is very efficient. We illustrate the program completion using an example from [81].

Example 2.2.2

$$\begin{aligned} p(d) &\leftarrow q(a), \sim q(b). \\ p(d) &\leftarrow q(b), \sim q(c). \\ q(a) &\leftarrow p(d). \\ q(b) &\leftarrow q(a). \end{aligned}$$

The Clark completion combines the rules for p into one rule, the rules for q into another rule, and replaces the $\leftarrow$ with $\leftrightarrow$. After some simplifications,

$$\begin{aligned} p(d) &\leftrightarrow (q(a) \wedge \neg q(b)) \vee (q(b) \wedge \neg q(c)) \\ \forall X(q(X) &\leftrightarrow ((X = a) \wedge p(d)) \vee ((X = b) \wedge q(a)) \end{aligned}$$

While the program completion approach is computationally very attractive (PROLOG for instance, uses SLDNF resolution) it has a few drawbacks. One of the major drawbacks

is that one some examples, the interpreter fails to derive conclusions when a goal neither succeeds nor fails. In some cases, the program completion is inconsistent. Also sometimes even when the completion is consistent, the conclusions are not very intuitive. The following example from [81] demonstrates that problem.

Example 2.2.3

$$
\begin{aligned}
b(1,2) &\leftarrow \\
b(2,1) &\leftarrow \\
g(2,3) &\leftarrow \\
g(3,2) &\leftarrow \\
p(X,Y) &\leftarrow b(X,Y). \\
p(X,Y) &\leftarrow b(X,U), p(U,Y). \\
e(X,Y) &\leftarrow g(X,Y). \\
e(X,Y) &\leftarrow g(X,U), e(U,Y). \\
a(X,Y) &\leftarrow e(X,Y), \sim p(X,Y).
\end{aligned}
$$

In this example it is easy to see that p is the transitive closure of b and e is the transitive closure of g. The predicate a is the difference between e and p. It appears that $a(2,3)$ should be true. However the program completion also admits models in which $p(2,3)$ and $p(1,3)$ are true and $a(2,3)$ is false.

2.2.2 3-valued program completion

The landmark paper of Fitting [29] introduced a semantics for logic programs with negation that was based on the three-valued logic of Kleene. The most important feature of this semantics, which we will henceforth call the Fitting semantics, is that every logic program

has a unique least model. Fitting's semantics was based on the notion of partial interpretations. We briefly describe here the Fitting semantics. The reader is referred to [29] for a detailed exposition.

Definition 1. *A partial interpretation is a pair* $\langle I^+, I^- \rangle$*, where* I^+ *and* I^- *are any subsets of the Herbrand base.*

A partial interpretation I is consistent if $I^+ \cap I^- = \emptyset$. For any partial interpretations I and J, we let $I \cap J$ be the partial interpretation $\langle I^+ \cap J^+, I^- \cap J^- \rangle$, and $I \cup J$ be the partial interpretation $\langle I^+ \cup J^+, I^- \cup J^- \rangle$. We also say that $I \subseteq J$ whenever $I^+ \subseteq J^+$ and $I^- \subseteq J^-$. While the collection of all consistent partial interpretations is closed under arbitrary intersections, it is not closed under arbitrary unions. Therefore the collection, under the subset relation $\subseteq$, does not form a complete lattice, but can be seen to meet the conditions for the following weaker structure:

Definition 2. $\langle S, \subseteq \rangle$ *is a complete semilattice if*

1. *The set* S *is partially ordered by* $\subseteq$
2. *Every nonempty subset of* S *has a greatest lower bound in* S *and*
3. *every nonempty directed subset of* S *has a least upper bound in* S *(A subset* A *is directed if for every* $X, Y \in A$ *there is some* $Z \in A$ *such that* $X \subseteq Z$ *and* $Y \subseteq Z$*)*

Complete semilattices are weaker structures than complete lattices, and monotonic maps on them are guaranteed to possess only unique least fixed points, but not greatest fixed points. For our purposes, however, this property is sufficient.

Definition 3. *Let* S *be partially ordered by* $\subseteq$*. A map* $T : S \to S$ *is monotonic, if for any* $X, Y \in S$*,* $X \subseteq Y$ *implies* $T(X) \subseteq T(Y)$*.*

The Fitting model of a general logic program P is the least fixed point of the immediate consequence function T_P^F on consistent partial interpretations defined as follows (let P^* the ground version of P):

Definition 4. *Let I be a partial interpretation. Then $T_P^F(I)$ is the partial interpretation given by*

$$\begin{aligned} T_P^F(I)^+ &= \{a \mid \textit{ for some clause } a \leftarrow l_1, l_2, \ldots, l_m \in P^*, \textit{ for each } 1 \leq i \leq m \\ &\quad \textit{if } l_i \textit{ is positive } l_i \in I^+ \textit{ and,} \\ &\quad \textit{if } l_i \textit{ is negative } l_i' \in I^-\} \\ T_P^F(I)^- &= \{a \mid \textit{ for every clause } a \leftarrow l_1, l_2, \ldots, l_m \in P^*, \textit{ there is some } 1 \leq i \leq m \\ &\quad \textit{if } l_i \textit{ is positive } l_i \in I^- \textit{ and,} \\ &\quad \textit{if } l_i \textit{ is negative } l_i' \in I^+\} \end{aligned}$$

where l_i' is the complement of the literal l_i.

It is easily seen that T_P^F is monotonic and its application on consistent partial interpretations results in a consistent partial interpretation. It thus possesses a least fixed point, which is the Fitting model for P. This least fixed point is easily shown to be $T_P^F \uparrow \omega$, where the ordinal powers of T_P^F are defined as follows:

Definition 5. *For any ordinal α,*

$$T_P^F \uparrow \alpha = \begin{cases} \langle \emptyset, \emptyset \rangle & \textit{if } \alpha = 0, \\ T_P^F(T_P^F \uparrow (\alpha - 1)) & \textit{if } \alpha \textit{ is a successor ordinal,} \\ \langle \cup_{\beta<\alpha}(T_P^F \uparrow \beta)^+, \ \cup_{\beta<\alpha}(T_P^F \uparrow \beta)^- \rangle & \textit{if } \alpha \textit{ is a limit ordinal.} \end{cases}$$

Example 2.2.4

Let P be the following general deductive database:

```
r(a,c)
r(b,b)
s(a,a)
```

$$\texttt{p(X)} \leftarrow \texttt{r(X,Y)}, \sim\texttt{p(Y)}$$
$$\texttt{p(Y)} \leftarrow \texttt{s(Y,a)}$$

Then, $T_P^F \uparrow 0 = \langle \emptyset, \emptyset \rangle$. $T_P^F \uparrow 1$ is the following partial interpretation:

$$(T_P^F \uparrow 1)^+ = \{ \texttt{r(a,c)}, \texttt{r(b,b)}, \texttt{s(a,a)} \},$$
$$(T_P^F \uparrow 1)^- = \{ \texttt{r(a,a)}, \texttt{r(a,b)}, \texttt{r(b,a)}, \texttt{r(b,c)}, \texttt{r(c,a)}, \texttt{r(c,b)}, \texttt{r(c,c)},$$
$$\texttt{s(a,b)}, \texttt{s(a,c)}, \texttt{s(b,a)}, \texttt{s(b,b)}, \texttt{s(b,c)},$$
$$\texttt{s(c,a)}, \texttt{s(c,b)}, \texttt{s(c,c)} \}.$$

And $T_P^F \uparrow 2 = I \cup T_P^F \uparrow 1$, where I is the partial interpretation $\langle \{\texttt{p(a)}\}, \{\texttt{p(c)}\} \rangle$. Furthermore, for every ordinal $\alpha > 2$, $T_P^F \uparrow \alpha$ can be seen to be the same as $T_P^F \uparrow 2$. Note that in the Fitting model the atom `p(a)` is *true* and the atom `p(c)` is *false*. No truth value is assigned to the atom `p(b)`.

The Fitting semantics has the distinction of being the first semantics to assign a unique model to general logic programs. However, it suffers from a drawback. It does not truly extend the van Emdem-Kowalski semantics definite logic programs. The Fitting semantics fails to capture *positive recursion*.

Example 2.2.5

Consider the following logic program:

$$a(0) \leftarrow b(0).$$
$$b(0) \leftarrow a(0).$$

The Fitting model of the program is $\langle \emptyset, \emptyset \rangle$. Thus both $a(0)$ and $b(0)$ are assigned the truth value *unknown*. But since this is a definite logic program, the van Emden-Kowalski semantics declares both $a(0)$ and $b(0)$ to be *false*. It is easy to see that there is a positive

recursion between the atoms $a(0)$ and $b(0)$. The Fitting semantics thus does not extend the van Emden-Kowalski semantics.

2.2.3 The well-founded semantics

Arguably the most widely accepted semantics for general logic programs is the well-founded semantics of van Gelder et al. The well-founded semantics extends the van Emden-Kowalski semantics to general logic programs. The reader is referred to [81] for a detailed description of the well-founded semantics.

The well-founded semantics is a 3-valued semantics. The negative conclusions in a general logic program are derived in the well-founded semantics on the basis of *unfounded sets*. We first define unfounded sets.

Definition 6. *A set $A \subseteq HB_P$ is an unfounded set of a general logic program P with respect to a partial interpretation I if each atom $p \in A$ satisfies the following condition: For each rule R of P whose head is p, atleast one of the following hold:*

1. *Some positive subgoal q of the body is false in I*

2. *Some positive subgoal of the body occurs in A*

We illustrate unfounded sets through an example from [81].

Example 2.2.6

Consider the following ground logic program

$$
\begin{aligned}
p(a) &\leftarrow p(c), \sim p(b).\\
p(b) &\leftarrow \sim p(a).\\
p(e) &\leftarrow \sim p(d).\\
p(c) &\leftarrow .\\
p(d) &\leftarrow q(a), \sim q(b).\\
p(d) &\leftarrow q(b), \sim q(c).\\
q(a) &\leftarrow p(d).\\
q(b) &\leftarrow q(a).
\end{aligned}
$$

The atoms $\{p(d), q(a), q(b), q(c)\}$ form an unfounded set with respect to the interpretation $\emptyset$. $q(c)$ satisfies the first condition and the other three atoms satisfy the second condition. It can be seen that $p(d), q(a)$ and $q(b)$ depend positively on each other. As a result, none of them can be the first to be proven true. Also declaring any one of them false does not make any of the remaining two true. This is where the set $\{p(a), p(b)\}$ does not form an unfounded set even though they depend on each other. The dependence is through negation. As a result, making one of them false results in the other being declared true.

Simultaneously negating all atoms in the unfounded set generalizes negation by failure in horn clause programs. If H is the Herbrand base of a Horn clause program and I is its least Herbrand model, then the atoms in $H - I$ form an unfounded set with respect to I.

It is easily seen that the arbitrary union of unfounded sets is an unfounded set.

Definition 7. *The greatest unfounded set of P with respect to I, denoted $GUS_P(I)$, is the union of all sets unfounded with respect to I.*

We are now ready to define the well-founded partial model of P. We first define three monotonic transformations.

Definition 8. *Transformations T_P, GUS_P and W_P are defined as follows:*

- *$p \in T_P(I)$ if and only if there is some instantiated rule R in P such that R has head p, and each subgoal literal in the body of R is true in I*
- *$GUS_P(I)$ is the greatest unfounded set of P with respect to I, as in Definition 7*
- *$W_P(I) = T_P(I) \cup \neg.GUS_P(I)$*

where $\neg.S$ of a set of atoms S is the set of all the complementary literals of the atoms in S.

Definition 9. *The well-founded semantics of a program P is the least fixed point of W_P. Every positive literal denotes that its atom is true, every negative literal denotes that its atom is false, and missing atoms have no truth value assigned to them in the semantics.*

2.2.4 Stable model semantics

We next define an important semantics for general logic programs that markedly differs from the semantics described so far. The stable model semantics was defined by Gelfond and Lifschitz in [32]. It has its roots in a nonmonotonic reasoning formalism called autoepistemic logic [56]. A notable feature of the stable model semantics is its simplicity.

The stable model semantics is a 2-valued semantics. We first define the stable models of a logic program without negation i.e., the definite logic programs.

Definition 10. *The least model of a definite logic program (without the appearance of $\sim$) is the smallest set of atoms M such that for every rule of the form*

$$A \leftarrow B_1, \ldots, B_n$$

if $B_1, \ldots, B_n \in M$, then $A \in M$.

This definition is the same as the T_P for definite logic programs defined by van Emden and Kowalski. For general logic programs, the stable model is based on a set of atoms. We assume that a set of atoms is available to us and based on a certain transformation to be defined, we decide whether this set is a stable model or not.

Definition 11. *Let P be a ground general logic program and let S be a set of atoms. The Gelfond-Lifschitz transformation P^S of P w.r.t S is obtained by*

1. *Deleting every rule with $\sim L$ in the body with $L \in S$*

2. *Deleting the negative literals from the bodies of the remaining rules*

P^S is a definite logic program. S is a stable model of P iff S is the least model of P^S.

It can be seen that this definition is simple and elegant. However, the stable model semantics is not constructive and hence is computationally expensive. It is apparent that a general logic program can have a number of stable models. The semantics of the program is taken to be the set of atoms in the intersection of all the stable models.

Example 2.2.7

Consider the ground program

$$a \leftarrow \sim b.$$
$$b \leftarrow \sim a.$$

This program has two stable models $\{a\}$ and $\{b\}$.

There is a difference between the stable model semantic and the other semantics discussed so far. While the well-founded semantics is a *skeptical semantics*, which means that the well-founded conclusions are only those that are necessarily true, the stable model semantics is a *credulous semantics*. Each stable model corresponds to a possible set of beliefs. Thus, when a program has more than one stable model, it essentially means that there is more than one way in which the meaning of the program may be interpreted.

There is a close connection between the well-founded and stable models of a program.

Theorem 2.2.1. *If the well-founded model of a program P is total, then it is the unique stable model for P.*

However, as shown in [81], the converse is not true. There are programs with unique stable models which do not coincide with the well-founded model.

Example 2.2.8

$$
\begin{aligned}
a &\leftarrow \sim b.\\
b &\leftarrow \sim a.\\
p &\leftarrow \sim p.\\
p &\leftarrow \sim b.
\end{aligned}
$$

This program has a unique stable model $\{a, p\}$ whereas its well-founded model is empty.

2.2.5 An alternative definition of the well-founded semantics

The well-founded semantics may be defined in terms of the Gelfond-Lifschitz transform. This was first demonstrated by Baral and Subrahmaniam in [8]. Let us denote the Gelfond-Lifschitz transform by Γ_P, i.e., Γ_P is a map on interpretations. For a set of atoms S, $\Gamma_P(S)$ is simply the least model of P^S.

It was shown in [8] that the well-founded positive conclusions of the general logic program P can be obtained by iterating Γ_P^2 from below. i.e., the least fixed point of Γ_P^2 is the set of well-founded positive conclusions. The fact that Γ_P^2 possesses a least fixed point follows from the observation that Γ_P is anti-monotonic. As a result, Γ_P^2 is monotonic and hence possesses a least fixed point. Let us denote by $(\Gamma_P^2)^{fp}(P)$ the least fixed point of Γ_P^2. Then $\Gamma((\Gamma_P^2)^{fp}(P))$ gives the set of conclusions that are possibly true. Hence $HB_P - \Gamma((\Gamma_P^2)^{fp}(P))$ is the set of all negative well-founded conclusions. It can be seen that Γ_P alternates in a certain fashion. That is, one application of Γ_P produces conclusions that are possibly true and a second application produces conclusions that are necessarily true. Thus even powers

of Γ_P produce conclusions that are necessarily true. Hence the least fixed point of Γ_P^2 is the set of well-founded positive conclusions. This was first demonstrated by van Gelder in [80].

2.2.6 Stratified negation

The search for a canonical model has often been targeted at identifying the class of programs that it can interpret satisfactorily. The least controversy surrounds a class of logic programs called *stratified programs*.

A program is stratified if its predicates can be assigned a partial ordering priority relation $<$ such that

1. Negative premises must have higher priority than heads
2. Positive premises must have priority greater than or equal to that of the heads

For the class of stratified (and locally stratified) programs, Przymusinski gives the perfect model semantics We describe this notion in terms of the description in [63].

This notion can be formalized through a *dependency graph* G for a program P whose vertices are the predicate symbols. For any two predicate symbols A and B, there is an edge from A to be B iff there is a clause in P such that B is the head of the clause and A is in the body. If B is negative, then we draw a negative edge.

We can now define priority relations $\leq$ and $<$ on the set of all predicate symbols in program P. $A \leq B$ (resp. $A < B$) if there is a directed path in G leading from B to A (resp. passing through atleast one negative edge).

Example 2.2.9

$$\begin{aligned} a(X) &\leftarrow b(X), \sim g(X). \\ g(X) &\leftarrow p(X). \end{aligned}$$

Here we have $a < g$ and $a \leq b, g \leq p$.

The goal now is to minimize the extensions of higher priority predicates as much as possible. Does is done even at the expense of increasing the extensions of lower priority predicates. Thus if M is a model of P and a new model N is obtained by changing some predicates, we say that N is *preferable* to M iff the addition of new elements to the extension (in M) of a lower priority predicate A is justified by removing elements from the extension (in M)of a higher priority predicate B.

Definition 12. *Let M and N be two models of a program P. Denote by $E_M(A)$ and $E_N(A)$ the extensions of a predicate A of P in M and N respectively. We say N is preferable to M, $N \prec M$, if for every predicate A such that $E_N(A) - E_M(A)$ is non-empty there is a predicate $B > A$ such that $E_M(B) - E_N(B)$ is non-empty. A model N is a perfect model of P if there are no models preferable to N.*

This definition can be extended by defining the priority relation $<$ on the set of all *ground atoms* instead of predicate symbols. When the extended $<$ is a partial order, then the program is called *locally stratified*.

The program in example 2.2.7 has two minimal models $M_1 = \{a\}$ and $M_2 = \{b\}$ but since $a \prec b$ and $b \prec a$ we have $M_1 \prec M_2$ and $M_2 \prec M_1$ and neither of them is perfect.

The following result is from [81].

Theorem 2.2.2. *If P is locally stratified, then it has a well-founded model, which is identical to the perfect model.*

2.2.7 Beyond stratification

It was thought that programs that are not stratified (locally stratified) do not make "sense". However it was shown by Kolaitis in [43] that there are queries in fixpoint logic that are not expressible by stratified programs. Consider the following program which describes a game in which a player wins when there is no move for the opponent, as in checkers.

Example 2.2.10

$$win(X) \leftarrow move(X, Y), \sim win(Y)$$

This program says that a player wins the game if the board is in position X and the move results in position Y and Y is not a winning position. This program is not locally stratified because the Herbrand instantiation of the program contains the rule

$$win(a) \leftarrow move(a, a), \sim win(a)$$

Thus even when the EDB does not contain a cycle of this form, the perfect model of the program is destroyed. However, Przymusinska and Przymusinski define the *weakly perfect models* which handle such a situation.

In order to efficiently compute semantics for such non-stratified databases, Ross introduced the concept of *modularly stratified programs* [65]. In order for a logic program and data to be modularly stratified, it should be possible to divide the predicates into modules with the following properties:

1. It is possible to order the modules so that predicates in a module depend only on predicates in that or previous modules.

2. Each module has a locally stratified model when we instantiate its rules and treat subgoals in previous modules as true or false according to the well-founded model for its module.

This is illustrated by an example in [77]. Consider a variation of the checkers program in which we introduce a new predicate $move1$ which is simply a copy of $move$.

Example 2.2.11

$$\begin{aligned} win(X) &\leftarrow move1(X,Y), \sim win(Y) \\ move1(X,Y) &\leftarrow move(X,Y). \end{aligned}$$

In this case, even though the program is not locally stratified if *move* is acyclic, the rules are modularly stratified. Because *move* is acyclic, we can put both *move* and *move*1 in the same module. Now since *move*1 is simply a copy of *move*, it behaves like an EDB predicate, and so we can compute the locally stratified model for this module even though *move* is acyclic.

2.3 Other Semantics

Although most researchers are convinced that the well-founded and stable model semantics are in a sense the final frontier, there has been some skepticism. There are examples of programs for which neither the well-founded nor the stable model semantics derive intuitive conclusions. Such examples are handled by other semantics that deserve special mention. One of them is the *stable class* semantics of Baral and Subrahmanian [9], which is an extension of the stable model semantics. Essentially, it assigns a set of stable models as the semantics of the program. This set is closed under the Γ operator introduced earlier.

Another direction of research has been by applying argumentation based semantics. Perhaps the most important work in this area is that of Dung [25]. This work generalizes both the well-founded and stable model semantics. The general idea is to consider sets of negated literals, called *hypothesis*.

Definition 13. *A scenario of a logic program P is a consistent theory $P \cup H$, where H is a set of negated literals.*

Definition 14. *A set of hypotheses E is called an evidence of an atom A if $P \cup E \vdash A$*

Definition 15. *A hypothesis $\sim A$ is said to be acceptable with respect to a scenario S if for every evidence E of A, there is $\sim B \in E$ such that $S \vdash B$*

Definition 16. *A scenario $S = P \cup H$ is admissible if each hypothesis $\sim A \in H$ is acceptable w.r.t S*

Definition 17. *A preferred extension of a program P is a maximal admissible scenario of P*

It was shown in [25] that both the stable models and the well-founded models can be described in terms of the preferred extensions.

Other developments are the three-valued stable model semantics of Przymusinski [62] and the partial stable models of Sacca and Zaniolo [68]. It has been shown that the partial stable models are the preferred extensions.

CHAPTER 3.

CONSTRAINTS AND REPAIRS

In this chapter we introduce the notion of constraints on a relational database and an associated problem: the state in which a database fails to satisfy the constraints imposed on it. We assume that the reader is somewhat familiar with the theory of dependencies in a relational database. The reader is referred to [27] and [1] for a detailed study of constraints. We present a broad overview of the constraints typically seen in a database and present recent developments in handling databases that do not obey constraints imposed on them.

The relational model, as defined by Codd [22], is as such devoid of any *semantics*. The tuples in a relation simply correspond to data values that are related to each other. The exact nature of this relationship is not apparent. For instance, whether the relationships are one-to-one, many-to-one and so on is not implicit in the model. However, Codd himself introduced the notion of constraints in a database. the first such constraint to be introduced is the *functional dependency*.

3.1 Constraints

This section introduces a number of constraints commonly seen in a database.

3.1.1 Functional dependency

Consider the following database taken from [27]. It describes a database with attributes $\{EMP, DEPT, MGR\}$.

This relation obeys the functional dependency(or FD) $DEPT \rightarrow MGR$ read as "$DEPT$ determines MGR". This means that whenever two tuples agree in the $DEPT$ column, they must also agree on the MGR column. More formally, let X and Y be subsets of the set of

EMP	DEPT	MGR
Hilbert	Math	Gauss
Pythagoras	Math	Gauss
Turing	Computer Science	von Neumann

Figure 3.1. An employee relation

attributes. Then FD $X \rightarrow Y$ is said to hold for an instance of a relation I if every pair of tuples that agree on each of the attributes in X also agree in the attributes in Y.

It is easy to see that FDs can be represented as sentences in first order logic. For example, if we were dealing with a 4-ary relation where the first, second, third and fourth columns are called A, B, C and D respectively, then the FD $AB \rightarrow C$ can be represented by the following sentence:

$$(\forall abc_1c_2d_1d_2)((Pabc_1d_1 \wedge Pabc_2d_2) \rightarrow (c_1 = c_2)) \tag{3.1}$$

3.1.2 Multivalued dependency

The next dependency to be introduced was the *multivalued dependency*(or MVD) discovered independently by Fagin and Zaniolo. There was a perception that the functional dependency did not completely capture the notion of "depends on". Consider the following relation, again from [27].

EMP	SALARY	CHILD
Hilbert	\$80K	Hilda
Pythagoras	\$30K	Peter
Pythagoras	\$30K	Paul
Turing	\$70K	Tom

Figure 3.2. An employee-salary-child relation

It can be seen that this instance obeys the FD $EMP \rightarrow SALARY$. however it does not obey the FD $EMP \rightarrow CHILD$. But it is obvious that an employee "determines" his children

in some sense. It is this notion that a multivalued dependency captures. we shall see that the multivalued dependency $EMP \twoheadrightarrow CHILD$ holds for this instance. as another example, consider the following relation with attributes EMP, $CHILD$ and $SKILL$.

EMP	CHILD	SKILL
Hilbert	Hilda	Math
Hilbert	Hilda	Physics
Pythagoras	Peter	Math
Pythagoras	Paul	Math
Pythagoras	Peter	Philosophy
Pythagoras	Paul	Philosophy
Turing	Tom	Computer Science

Figure 3.3. An employee-child-skill relation

It can be seen that this relation does not obey any functional dependencies. However, it obeys the MVDs $EMP \twoheadrightarrow CHILD$ and $EMP \twoheadrightarrow SKILL$.

We can now formally define MVDs. Let I be a relation over a scheme U. Let X,Y be subsets of U and let $Z = U - XY$. The multivalued dependency $X \twoheadrightarrow Y$ holds for the relation I if for every pair of tuples r and s for which $r[X] = s[X]$, there is a tuple t in I such that $t[X] = r[X] = s[X]$ and $t[Y] = r[Y]$ and $t[Z] = s[Z]$. It follows by symmetry that there is also a tuple u in I such that $u[X] = r[X] = s[X]$ and $u[Y] = s[Y]$ and $u[Z] = r[Z]$.

Like FDs, MVDs can also be expressed in first order logic. For example, assume $U = \{A, B, C, D, E\}$. Then the MVD $AB \twoheadrightarrow CD$ holds for a relation over U if the following sentence holds, where P is the relation symbol.

$$(\forall abc_1c_2d_1d_2e_1e_2)((Pabc_1d_1e_1 \wedge Pabc_2d_2e_2) \rightarrow Pabc_2d_2e_1) \tag{3.2}$$

3.1.3 Join dependency

The join dependency may be defined as follows. Let $\mathbf{X} = \{X_1, \ldots, X_k\}$ be a collection of subsets of U such that $U = X_1 \cup \ldots \cup X_k$. The relation I over U is said to obey

the join dependency $\bowtie [X_1, \ldots, X_k]$, denoted by $\bowtie [\mathbf{X}]$, if I is the join of its projections $I[X_1], \ldots, I[X_k]$. It follows that the join dependency holds for the relation I if and only if I contains each tuple t for which there are tuples $w_1, \ldots, w_n$ of I such that $w_i[X_i] = t[X_i]$ for each $1 \leq i \leq n$. Consider the following relation.

A	B	C	D
0	1	0	0
0	2	3	4
5	1	3	0

Figure 3.4. Join dependency

This relation violates the join dependency $\bowtie [AB, ACD, BC]$. Let w_1, w_2 and w_3 be the tuples $(0, 1, 0, 0), (0, 2, 3, 4)$ and $(5, 1, 3, 0)$ respectively. Let X_1, X_2, X_3 respectively be AB, ACD and BC. Let t be the tuple $(0, 1, 3, 4)$. Now $w_i[X_i] = t[X_i]$ for each $1 \leq i \leq n$ although t is not in I. However this relation obeys the join dependency $\bowtie [ABC, BCD, ABD]$.

Join dependencies can also be written in first order logic. if we are dealing with relations with attirbutes c, t, r, s, h, g then the join dependency $\bowtie [TC, CRH, SGC]$ can be written as

$$(\forall ct_1t_2rr_1r_2hh_1h_2ss_1s_2gg_1g_2)((Pctr_1h_1s_1g_1 \wedge Pctr_1h_1s_2g_2 \wedge Pct_2r_2h_2sg) \rightarrow Pctrhsg) \quad (3.3)$$

3.1.4 Inclusion dependency

The dependencies that we have discussed so far can be seen to have the following two properties:

1. they are *uni-relational*

2. they are typed

By uni-relational we mean that they deal with one relation at a time. By typed we mean that no variables appear in two distinct columns. For example, the sentence $(\forall xy)((Pxy \wedge Pyz) \rightarrow$

Pxz), which says that a relation is transitive, is not typed since y appears in both the first and second columns of P. Inclusion dependencies (or IND) violate both of these properties. For example, an IND can say that every manager entry of the P relation appears as an employee in the Q relation. In general an IND is of the form

$$P[A_1, \ldots, A_m] \subseteq Q[B_1, \ldots, B_m] \tag{3.4}$$

where P and Q are relation names(possibly the same), where the A_i's and B_i's are attributes. If I is the P relation and J is the Q relation, then the IND 3.4 holds if for each tuple s of I, there is a tuple t of J such that $s[A_1, \ldots, A_m] \subseteq t[B_1, \ldots, B_m]$. INDs are commonly referred to as ISA relationships in the AI community. INDs can also be expressed in first order logic. For example, if the P relation has attributes ABC, and the Q relation has the attributes CDE, then the IND $P[AB] \subseteq Q[CE]$ can be written as

$$(\forall abc)(Pabc \rightarrow \exists dQadb) \tag{3.5}$$

It was observed that all the dependencies discussed above could united into a single class, called *dependencies*. A *dependency* is a first order sentence

$$(\forall x_1, \ldots, x_m)((A_1 \wedge \ldots \wedge A_n) \rightarrow \exists y_1, \ldots, y_r(B_1 \wedge \ldots \wedge B_s)) \tag{3.6}$$

We assume that each of the x_j's appears in atleast one of the A_i's, and that $n \geq 1$. We assume that $r \geq 0$ and that $s \geq 1$. It can be seen that the empty database obeys these constraints. Also, we can check whether a dependency holds for a relation simply by assuming that the quantifiers range over the elements that appear in the relation, a property called *domain independence*.

If each formula B_i on the right-hand side of equation 3.6 is a relational formula, the dependency is called a *tuple generating dependency*; if all of these formulas are equalities,

then we call the dependency an *equality generating dependency*. Of the dependencies we have seen, multi-valued dependencies, join dependencies and inclusion dependencies are tuple generating dependencies and functional dependencies are equality generating dependencies. When $r = 0$ and $s = 1$ in Equation 3.6 the dependency is called a *full dependency*. Thus, a full dependency is of the form

$$(\forall x_1, \ldots, x_m)((A_1 \wedge \ldots \wedge A_n) \rightarrow B) \tag{3.7}$$

where each A_i is a relational formula and B is atomic. Functional, multi-valued and join dependencies are full dependencies. Note that allowing several atomic formulas on the right-hand side results in no gain since such a sentence is equivalent to a finite set of full dependencies.

It should be noted that in the "real world" dependencies rarely appear in their most general form. FDs, INDs and maybe MVDs are the only kinds of dependencies that may be called "real world dependencies".

3.1.5 Conditional functional dependencies

Conditional functional dependencies are a recent development. These were introduced by Fan et al with the aim of capturing constraints in data for which the traditional FDs incur high complexity. The reader is referred to [28] for a detailed description. Here we provide a brief introduction to it since it plays an important role in the problem of data cleaning.

We first illustrate conditional functional dependencies (or CFDs) through an example. This example is from [28]. Consider a relation schema cust(CC, AC, PN, NM, STR, CT,ZIP), which specifies a customer in terms of the customers phone (country code(CC), area code (AC), phone number (PN)), name (NM), and address (street (STR), city (CT), zip code (ZIP)). Figure 3.1.5 is an instance of this relation. Traditional FDs on this relation include:

CC	AC	PN	NM	STR	CT	ZIP
01	908	1111111	Mike	Tree Ave.	NYC	07974
01	908	1111111	Rick	Tree Ave.	NYC	07974
01	212	2222222	Joe	Elm St.	NYC	01202
01	212	2222222	Jim	Elm St.	NYC	02404
01	215	3333333	Ben	Oak Ave.	PHI	02394
44	131	4444444	Ian	High St.	EDI	EH4 1DT

Figure 3.5. A CFD illustration

$$\begin{aligned} f_1 : [CC, AC, PN] &\rightarrow [STR, CT, ZIP] \\ f_2 : [CC, AC] &\rightarrow [CT]. \end{aligned}$$

Now consider the following constraint on this relation. In the UK, ZIP determines STR. This can be expressed as

$$\phi_0 : [CT = 44, ZIP] \rightarrow [STR]$$

In other words, ϕ_0 is an FD that is to hold on the subset of tuples that satisfies the pattern "CC = 44", rather than on the entire cust relation. It is generally not considered an FD in the standard definition since ϕ_0 includes a pattern with data values in its specification.

$$\begin{aligned} \phi_1 : [CC = 01, AC = 908, PN] &\rightarrow [STR, CT = MH, ZIP] \\ \phi_2 : [CC = 01, AC = 212, PN] &\rightarrow [STR, CT = NYC, ZIP] \\ \phi_3 : [CC = 01, AC = 215] &\rightarrow [CT = PHI]. \end{aligned}$$

Constraint ϕ_1 assures that only in the US (country code 01) and for area code 908, if two tuples have the same PN, then they must have the same STR and ZIP and moreover, the city must be MH. Similarly,ϕ_2 assures that if the area code is 212 then the city must be NYC; and ϕ_3 specifies that for all tuples in the US and with area code 215, their city must be PHI (irrespective of the values of the other attributes).

Observe that ϕ_1 and ϕ_2 refine the standard FD f1 given above, while ϕ_3 refines the FD $f2$. These refinements essentially enforce bindings of semantically related data values. Indeed, while tuples t1 and t2 in Figure 3.1.5 do not violate $f1$, they violate its refinement ϕ_1, since the city cannot be NYC if the area code is 908.

We now formally define CFDs. Consider a relation schema R defined over a fixed set of attributes, denoted by $attr(R)$. For each attribute $A \in attr(R)$, its domain is specified in R, denoted as $dom(A)$.

Definition 18. *A CFD φ on R is a pair $(R : X \rightarrow Y, T_p)$ where*

1. *X, Y are sets of attributes in $attr(R)$*
2. *$X \rightarrow Y$ is a standard FD, called the FD embedded in φ and*
3. *T_p is a tableau containing attributes in X and Y, called the pattern tableau of φ, where for each $A \in X \cup Y$ and each tuple $t \in T_p$, $t[A]$ is either a constant 'a' in $dom(A)$ or an unnamed variable '-' that draws values from $dom(A)$.*

If A occurs in both X and Y , we use $t[A_L]$ and $t[A_R]$ to indicate the occurrence of A in X and Y, respectively, and separate the X and Y attributes in a pattern tuple with '||'.

Example 3.1.1

The constraints $\phi_0, f_1, \phi_1, \phi_2, f_2$ and ϕ_3 can be expressed as CFDs φ_1 (for ϕ_0) φ_2 (for f_1, ϕ_1 and ϕ_2, one per line respectively) and φ_3 (for f_2, ϕ_3 and an additional $[CC = 44, AC = 141] \rightarrow [CT = GLA]$) as shown in Figure 3.6.

W now describe the semantics for CFDs. An instantiation ρ of a tableau tuple t_p in tableau T_p is a mapping from t_p to data values with no variables such that for each attribute $A \in X \cup Y$, if $t_p[A]$ is '-' ρ maps $t_p[A]$ to a constant in $dom(a)$ and if $t_p[A]$ is constant a, ρ maps it to the same constant a. For example, for $t_p[A, B] = (a, -)$ we can define ρ such that $t_p[A, B] = (a, b)$ which maps $t_p[A]$ to itself and $t_p[B]$ to a value $b \in dom(B)$.

Tableau T_1 for $\varphi_1 = ([CC, ZIP] \rightarrow [STR], T_1)$

CC	ZIP	STR
44	-	-

Tableau T_2 for $\varphi_2 = ([CC, AC, PN] \rightarrow [STR, CT, ZIP], T_2)$

CC	AC	PN	STR	CT	ZIP
-	-	-	-	-	-
01	908	-	-	MH	-
01	212	-	-	NYC	-

Tableau T_3 for $\varphi_3 = ([CC, AC] \rightarrow [CT], T_3)$

CC	AC	CT
-	-	-
01	215	PHI
44	141	GLA

Figure 3.6. An example of CFDs

A data tuple t is said to *match* a pattern tuple t_p, dnoted by $t \asymp t_p$ if there is an instantiation ρ such that $\rho(t_p) = t$.

An instance I of R satisfies a CFD φ, denoted by $I \models \varphi$, if for each pair of tuples $t_1, t_2 \in I$, and for each pattern tuple t_p in the pattern tableau T_p of φ, if $t_1[X] = t_2[X] \asymp t_p[X]$, then $t_1[Y] = t_2[Y] \asymp t_p[Y]$. That is, if $t_1[X]$ and $t_2[X]$ are equal and in addition, they both match the pattern $t_p[X]$, then $t_1[Y]$ and $t_2[Y]$ must also be equal to each other and both match the pattern $t_p[Y]$.

If Σ is a set of CFDs, we write $I \models \Sigma$ if $I \models \varphi$ for each $\varphi \in \Sigma$.

Example 3.1.2

The cust relation of Figure 3.1.5 satisfies φ_1 and φ_3 but violates φ_2. The first and second tuples violate the pattern tuple $t_p = (01, 908, - \,||\, -, MH, -)$.

It can be observed that while violation of FDs requires two tuples, a single tuple may violate a CFD.

Observe that a standard FD $X \rightarrow Y$ can be expressed as a CFD $(X \rightarrow Y, T_p)$ in which T_p contains a single tuple consisting of - only, without constants.

3.2 Repairs

The second half of this chapter focuses on the problem of query answering in a database that violates constraints imposed on it. The problem is commonly referred to as the problem of *consistent query answering* (or CQA). A database that violates constraints imposed on it is said to be *inconsistent*. Although the database is inconsistent, this inconsistency is usually local to a few tuples. A large portion of the information in the database remains useful. Thus being able to extract information from such databases is of significance. CQA addresses this problem. CQA has generally been approached from two directions: one is the *query rewriting* approach in which the original query is transformed into a new query that retrieves answers only from the consistent portion of the database. However, this approach can handle only a small class of constraints. The complexity of query rewriting gets extremely high for certain commonly seen constraints. The other approach is what is commonly known as *repairing* the database. Informally, the idea is to perform updates on the database such that consistency is restored. Such updates can be performed in a number of ways: by deleting tuples, by inserting tuples, or by modifying values in the database. This section focuses on this approach to the CQA problem. We first introduce repairs. The idea of repairs was first introduced by Arenas et al. in [3]. The reader is referred to this work for details.

We first demonstrate the inconsistency problem through an example. Assume that the Computer Science department of a university has more than one location and each location maintains the relation $Teaches(class, professor)$ of the class offered and the faculty member that teaches it. Assume that we have the functional dependency $class \rightarrow professor$ on this relation. The data is collected from two different locations and they are shown in Figure 3.7. Notice that each source by itself satisfies the integrity constraint but when combined the

Class	Professor
c1	p1
c2	p2
c3	p3

Class	Professor
c1	p2
c3	p3

Figure 3.7. Two relations whose union is inconsistent w.r.t FD

Class	Professor
c1	p1
c2	p2
c3	p3

Class	Professor
c1	p2
c2	p2
c3	p3

Figure 3.8. Minimal repairs of the $Teaches(Class, Professor)$ relation

constraint is violated. In this instance we have that course $c1$ is taught by both professors $p1$ and $p2$. However, as can be seen, such inconsistencies are local to a few tuples in the database and the rest of the database is consistent with the integrity constraints. In such a situation, it becomes important to be able to extract answers to queries from the consistent part of the database. A *repair* of a database is the set of changes made to the database so that consistency is restored. We are interested in the *minimal repairs*, the repairs that involve minimal updates to the orginal database (or maximal under set inclusion). A consistent query answer is defined as the set of tuples that is true in every minimal repair of the database. For the database shown in Figure 3.7 the minimal repairs are shown in Figure 3.8. We now formally define repairs.

Definition 19. *A relational database scheme R is a finite set of attribute names $\{A_1, \ldots, A_n\}$ where for any attribute name $A_i \in R$, $dom(A_i)$ is a non-empty set of constants denoting the domain of the attribute A_i.*

Definition 20. *A relational database instance r on a database scheme $R = \{A_1, \ldots, A_n\}$ is any subset of $dom(A_1) \times \ldots \times dom(A_n)$.*

A database scheme may be subjected to *integrity constraints*. A set of integrity constraints IC is a set of first order formulas over the database scheme which the database instance is expected to satisfy. If a database instance r satisfies IC in the standard model theoretic sense, then we say that r is *consistent* with IC, denoted by $r \models IC$. Otherwise, we say that r is *inconsistent*.

Example 3.2.1

Consider a database instance $r = \{p(a, b), p(b, c), p(a, c)\}$ subject to the functional dependency constraint $IC : p(X, Y) \wedge p(X, Z) \rightarrow Y = Z$. It is easy to see that $r \not\models IC$.

Example 3.2.2

Consider a database instance $r = \{p(a), p(b), q(a), q(c)\}$ subject to the constraint $IC : (\forall X)(p(X) \rightarrow q(X))$. $r \not\models IC$ since $q(b) \notin r$.

Definition 21. *Given a database instance r on a scheme R, and a set of integrity constraints IC, a repair of r w.r.t IC is a database instance r' on the scheme R, such that $r' \models IC$ and such that $r \Delta r' = (r - r') \cup (r' - r)$, the symmetric difference between r and r', is minimal under set inclusion.*

Example 3.2.3

The possible repairs of the database instance of Example 3.2.1 are $r_1 = \{p(a, b), p(b, c)\}$ and $r_2 = \{p(a, c), p(b, c)\}$

Example 3.2.4

The possible repairs of the database instance of Example 3.2.2 are $r_1 = \{p(a), p(b), q(a), q(b), q(c)\}$ and $r_2 = \{p(a), q(a), q(c)\}$

Definition 22. *A tuple t is a consistent answer to a query $Q(\bar{x})$ on a database instance r if t is an answer to Q on every repair r' of r w.r.t IC.*

$$Q(\bar{x}) = \{t \mid (\forall r')(r' \text{ is a repair of } r \rightarrow r' \models Q(t))\}$$

Example 3.2.5

The consistent answers to the query $q(X)$ on the database instance r of Example 3.2.2 are $\{q(a), q(c)\}$ since they appear in both repairs of r.

It is easy to see that the number of repairs can be exponential. The following example demonstrates such a situation.

Example 3.2.6

Consider the following relation subject to the FD $X \rightarrow Y$.

X	Y
a_1	b_1
a_1	b'_1
a_2	b_2
a_2	b'_2
...	
a_n	b_n
a_n	b'_n

Figure 3.9. A relation with an exponential number of repairs

It is easy to see that this relation has 2^n repairs.

3.2.1 Denial constraints

As shown in Figure 3.2.6 the complexity of CQA can be exponential even for relatively simple queries. It is then of interest to delineate the boundary between the queries that are tractable and those that are not. Chomicki and Marcinkowski in [19] consider *denial constraints*. The denial constraints are first order sentences of the form

$$\forall \bar{x}_1 \ldots \bar{x}_m \neg [P_1(\bar{x}_1) \wedge \ldots \wedge P_m(\bar{x}_m) \wedge \varphi(\bar{x}_1 \ldots \bar{x}_m)]$$

It is easy to see that FDs are a special case of denial constraints. It is shown in [19] that CQA is in PTIME for the following classes of queries and constraints.

1. queries: quantifier-free, constraints: arbitrary denial;

2. queries: simple conjunctive, constraints: functional dependencies (at most one FD per relation);

3. queries: quantifier-free or simple conjunctive, constraints: key functional dependencies and foreign key constraints, with at most one key per relation.

It can be seen that for our definition of repair, for the denial constraints repairs can be obtained by deletion alone. For such constraints the repairs can be represented by a *conflict hypergraph*.

Definition 23. *The conflict hypergraph $G_{F,r}$ is a hypergraph whose set of vertices is the set $\Sigma(r)$ of facts of an instance r and whose set of edges consists of all the sets*

$$\{P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots, P_l(\bar{t}_l)\}$$

such that

$$P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots, P_l(\bar{t}_l) \in \Sigma(r)$$

and there is a constraint

$$\forall \bar{x}_1 \ldots \bar{x}_l \neg [P_1(\bar{x}_1) \wedge \ldots \wedge P_l(\bar{x}_l) \wedge \varphi(\bar{x}_1 \ldots \bar{x}_l)]$$

such that $P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots, P_l(\bar{t}_l)$ together violate this constraint.

Note that there may be edges in $G_{F,r}$ that contain only one vertex. Also, the size of the conflict hypergraph is polynomial in the number of tuples in the database instance.

By an *independent set* in a hypergraph we mean a subset of its set of vertices which does not contain any edge.

We now state without proof a number of results on the complexity of CQA. The reader is referred to [19] for proofs.

Theorem 3.2.1. *For every set F of denial constraints and quantifier-free queries ϕ, CQA is in PTIME.*

Theorem 3.2.2. *Let F be a set of FDs, each dependency over a different relation among $P_1, P_2, \ldots, P_k$. Then for each closed simple conjunctive query Q, there exists a sentence Q' such that for every database instance r, $r \models_F Q$ iff $r \models Q'$. Consequently, CQA is in P.*

The above results are the strongest that can be obtained. For relaxing any of the above restrictions leads to co-NP-completeness.

Theorem 3.2.3. *There exist a key FD f and a closed conjunctive query*

$$Q \equiv (\exists x, y, z)(R(x, y, c) \wedge R(z', y, d) \wedge y = y')$$

for which CQA is co-NP-complete.

Theorem 3.2.4. *There is a set F of two key dependencies and a closed conjunctive query $Q \equiv (\exists x, y)(R(x, y, b))$, for which CQA is co-NP-complete.*

Theorem 3.2.5. *There exist a denial constraint f and a closed conjunctive query $Q \equiv (\exists x, y)(R(x, y, b))$, for which CQA is co-NP-complete.*

Theorem 3.2.6. *For every set of INDs I and query Q, CQA is in PTIME.*

The reader is referred to [19] for a number of complexity results on databases having both FD and IND constraints.

CHAPTER 4.
A PARACONSISTENT RELATIONAL DATA MODEL

In this chapter, we present some background material related to the thesis. We introduce a data model based on the OWA, called paraconsistent relations and an algebra for query processing on paraconsistent relations.

4.1 Paraconsistent Relations

In this section, we present a brief overview of a data model based on the OWA called paraconsistent relations and the algebraic operations on them. For a more detailed description, refer to [6]. Unlike ordinary relations that can model worlds in which every tuple is known to either hold a certain underlying predicate or to not hold it, paraconsistent relations provide a framework for incomplete or even inconsistent information about tuples. They naturally model *belief* systems rather than *knowledge* systems, and are thus a generalisation of ordinary relations. The operators on ordinary relations can also be generalised for paraconsistent relations.

4.2 Formal Definition of Paraconsistent Relations

Let a *relation scheme* (or just *scheme*) Σ be a finite set of *attribute names*, where for any attribute name $A \in \Sigma$, $dom(A)$ is a non-empty *domain* of values for A. A *tuple* on Σ is any map $t : \Sigma \to \cup_{A \in \Sigma} dom(A)$, such that $t(A) \in dom(A)$, for each $A \in \Sigma$. Let $\tau(\Sigma)$ denote the set of all tuples on Σ.

Definition 24. *A* paraconsistent relation *on scheme* Σ *is a pair* $R = \langle R^+, R^- \rangle$*, where* R^+ *and* R^- *are any subsets of* $\tau(\Sigma)$*. We let* $\mathcal{P}(\Sigma)$ *be the set of all paraconsistent relations on* Σ*.* □

Definition 25. *A paraconsistent relation R on scheme Σ is called a* consistent paraconsistent relation *if $R^+ \cap R^- = \varnothing$. We let $\mathcal{C}(\Sigma)$ be the set of all consistent relations on Σ. Moreover, R is called a* complete paraconsistent relation *if $R^+ \cup R^- = \tau(\Sigma)$. If R is both consistent and complete, i.e. $R^- = \tau(\Sigma) - R^+$, then it is a* total paraconsistent relation*, and we let $\mathcal{T}(\Sigma)$ be the set of all total paraconsistent relations on Σ.* □

Figure 4.1 is an example paraconsistent relation. The solid lines are used to separate tuples in a component and solid double lines are used to separate the positive and negative components.

supply

SNUM	PNUM
s1	p1
s1	p3
s2	p2
s3	p4
s1	p2
s2	p3
s3	p3

Figure 4.1. An example of a paraconsistent relation

4.3 Algebraic Operators on Paraconsistent Relations

We now define the relational algebra operators for paraconsistent relations. To reflect generalization of relational algebra operators, a dot is placed over an ordinary relational operator to obtain the corresponding paraconsistent relation operator. For example, $\bowtie$ denotes the natural join among ordinary relations, and $\dot{\bowtie}$ denotes natural join on paraconsistent relations. We first introduce two fundamental set-theoretic algebraic operators on paraconsistent relations:

Definition 26. *Let R and S be paraconsistent relations on scheme Σ. Then, the* union *of R and S, denoted $R \dot{\cup} S$, is a paraconsistent relation on scheme Σ, given by $(R \dot{\cup} S)^+ = R^+ \cup S^+$ and $(R \dot{\cup} S)^- = R^- \cap S^-$; and the* complement *of R, denoted $\dot{-} R$, is a paraconsistent relation on scheme Σ, given by $(\dot{-} R)^+ = R^-$ and $(\dot{-} R)^- = R^+$.* □

If Σ and Δ are relation schemes such that $\Sigma \subseteq \Delta$, then for any tuple $t \in \tau(\Sigma)$, we let t^Δ denote the set $\{t' \in \tau(\Delta) \mid t'(A) = t(A), \text{ for all } A \in \Sigma\}$ of all extensions of t. We extend this notion for any $T \subseteq \tau(\Sigma)$ by defining $T^\Delta = \cup_{t \in T}\ t^\Delta$. We now define some relation-theoretic operators on paraconsistent relations.

Definition 27. *Let R and S be paraconsistent relations on schemes Σ and Δ, respectively. Then, the* natural join *(or just* join*) of R and S, denoted $R \dot{\bowtie} S$, is a paraconsistent relation on scheme $\Sigma \cup \Delta$, given by $(R \dot{\bowtie} S)^+ = R^+ \bowtie S^+$, and $(R \dot{\bowtie} S)^- = (R^-)^{\Sigma \cup \Delta} \cup (S^-)^{\Sigma \cup \Delta}$, where $\bowtie$ is the usual natural join among ordinary relations.* □

Definition 28. *Let R be a paraconsistent relation on scheme Σ, and Δ be any scheme. Then, the* projection *of R onto Δ, denoted $\dot{\pi}_\Delta(R)$, is a paraconsistent relation on Δ, given by $\dot{\pi}_\Delta(R)^+ = \pi_\Delta((R^+)^{\Sigma \cup \Delta})$, and $\dot{\pi}_\Delta(R)^- = \{t \in \tau(\Delta) \mid t^{\Sigma \cup \Delta} \subseteq (R^-)^{\Sigma \cup \Delta}\}$, where π_Δ is the usual projection over Δ of ordinary relations.* □

Definition 29. *Let R be a paraconsistent relation on scheme Σ, and let F be any logic formula involving attribute names in Σ, constant symbols (denoting values in the attribute domains), equality symbol $=$, negation symbol $\neg$, and connectives $\vee$ and $\wedge$. Then, the* selection *of R by F, denoted $\dot{\sigma}_F(R)$, is a paraconsistent relation on scheme Σ, given by $\dot{\sigma}_F(R)^+ = \sigma_F(R^+)$, and $\dot{\sigma}_F(R)^- = R^- \cup \sigma_{\neg F}(\tau(\Sigma))$, where σ_F is the usual selection of tuples satisfying F.* □

4.4 Computing the Fitting Model Using Paraconsistent Relations

We now describe our method for computing the Fitting model for a given general deductive database P. In this model, partial relations are the semantic objects associated with the predicate symbols occurring in P.

Our method involves two steps. The first step is to convert P into a set of partial relation definitions for the predicate symbols occurring in P. These definitions are of the form

$$\mathtt{p} = D_{\mathtt{p}},$$

where $\mathtt{p}$ is a predicate symbol of P, and $D_{\mathtt{p}}$ is an algebraic expression involving predicate symbols of P and partial relation operators. The second step is to iteratively evaluate the expressions in these definitions to incrementally construct the partial relations associated with the predicate symbols.

In the remaining part of this section we describe our method to convert the given database P into a set of definitions for the predicate symbols in P. Before presenting the actual algorithm, let us look at an example. Consider the following program which contains only clauses with the predicate symbol p in their heads:

$$\begin{aligned} \mathtt{p(X)} &\leftarrow \mathtt{r(X,Y)},\ \neg\mathtt{p(Y)} \\ \mathtt{p(Y)} &\leftarrow \mathtt{s(Y,a)} \end{aligned}$$

From these clauses the algebraic definition constructed for the symbol p is the following:

$$\mathtt{p} = (\dot{\pi}_{\mathtt{\{X\}}}(\mathtt{r(X,Y)} \ \dot{\bowtie} \ \dot{-}\mathtt{p(Y)}))[\mathtt{X}] \ \dot{\cup} \ (\dot{\pi}_{\mathtt{\{Y\}}}(\dot{\sigma}_{\mathtt{Z=a}}(\mathtt{s(Y,Z)})))[\mathtt{Y}] \tag{4.1}$$

Such a conversion exploits the close connection between attribute names in relation schemes and variables in clauses, as pointed out by Ullman (1988).

The algebraic expression for the predicate symbol `p` is a union ($\dot{\cup}$) of the expressions obtained from each clause containing the symbol `p` in its head. It therefore suffices to give an algorithm for converting a clause into an expression.

Algorithm CONVERT

INPUT: A general deductive database clause $l_0 \leftarrow l_1, \ldots, l_m$. Let l_0 be an atom of the form $p_0(A_{01}, \ldots, A_{0k_0})$, and each l_i, $1 \leq i \leq m$, be a literal either of the form $p_i(A_{i1}, \ldots, A_{ik_i})$, or of the form $\neg p_i(A_{i1}, \ldots, A_{ik_i})$. For any i, $0 \leq i \leq m$, let V_i be the set of all variables occurring in l_i.

OUTPUT: An algebraic expression involving partial relations.

METHOD: The expression is constructed by the following steps:

1. For each argument A_{ij} of literal l_i, construct argument B_{ij} and condition C_{ij} as follows:

   (a) If A_{ij} is a constant a, then B_{ij} is any brand new variable and C_{ij} is $B_{ij} = a$.

   (b) If A_{ij} is a variable, such that for each k, $1 \leq k < j$, $A_{ik} \neq A_{ij}$, then B_{ij} is A_{ij} and C_{ij} is *true*.

   (c) If A_{ij} is a variable, such that for some k, $1 \leq k < j$, $A_{ik} = A_{ij}$, then B_{ij} is a brand new variable and C_{ij} is $A_{ij} = B_{ij}$.

2. Let $\hat{l}_i$ be the atom $p_i(B_{i1}, \ldots, B_{ik_i})$, and F_i be the conjunction $C_{i1} \wedge \cdots \wedge C_{ik_i}$. If l_i is a positive literal, then let Q_i be the expression $\dot{\pi}_{V_i}(\dot{\sigma}_{F_i}(\hat{l}_i))$. Otherwise, let Q_i be the expression $\dot{-}\dot{\pi}_{V_i}(\dot{\sigma}_{F_i}(\hat{l}_i))$.

   As a syntactic optimization, if all conjuncts of F_i are *true* (i.e. all arguments of l_i are distinct variables), then both $\dot{\sigma}_{F_i}$ and $\dot{\pi}_{V_i}$ are reduced to identity operations, and are henced dropped from the expression. For example, if $l_i = \neg$`p(X,Y)`, then $Q_i =$ $\dot{-}$`p(X,Y)`.

3. Let E be the natural join ($\dot{\bowtie}$) of the Q_i's thus obtained, $1 \leq i \leq m$. The output expression is $(\dot{\sigma}_{F_0}(\dot{\pi}_V(E)))[B_{01}, \ldots, B_{0k_0}]$, where V is the set of variables occurring in $\hat{l}_0$.

As in step 2, if all conjuncts in F_0 are *true*, then $\dot{\sigma}_{F_0}$ is dropped from the output expression. However, $\dot{\pi}_V$ is never dropped, as the clause body may contain variables not in V. □

From the algebraic expressions obtained by Algorithm **CONVERT** for clauses in the given general deductive database, we construct a system of equations defining partial relations as follows.

Definition 30. *For any general deductive database P,* EQN*(P) is a set of all equations of the form $\mathtt{p} = D_{\mathtt{p}}$, where* $\mathtt{p}$ *is a predicate symbol of P, and $D_{\mathtt{p}}$ is the union* $(\dot{\cup})$ *of all expressions obtained by Algorithm* **CONVERT** *for clauses in P with symbol* $\mathtt{p}$ *in their head. The algebraic expression $D_{\mathtt{p}}$ is also called a* definition *of* $\mathtt{p}$. □

Proposition 4.4.1. (Termination) *The above procedure for constructing EQN(P) terminates for any general deductive database P.*

Proof. Immediate from the fact that P has only finite number of clauses, each clause contains a finite number of literals, and each literal has a finite number of arguments. □ □

The second and final step in our model computation process is to incrementally construct the partial relations defined by the given database. For any general deductive database P, we let P_E and P_I denote its extensional and intensional portions, respectively. P_E is essentially the set of clauses of P with empty bodies, and P_I is the set of all other clauses of P. Without loss of generality, we assume that no predicate symbol occurs both in P_E and in P_I. Let us recall that $P_E^\star$ is the set of all ground instances of clauses in P_E.

The overall computation algorithm is rather straightforward. It treats the predicate symbols in a given database as imperative "variable names" that may contain a partial relation as value. Thus, any variable $\mathtt{p}$ has two set-valued fields, namely $\mathtt{p}^+$ and $\mathtt{p}^-$.

Algorithm COMPUTE

INPUT: A general deductive database P.

OUTPUT: Partial relation values for the predicate symbols of P.

METHOD: The values are computed by the following steps:

1. *(Initialization)*

   (a) Compute $EQN(P_I)$ using Algorithm **CONVERT** for each clause in P_I.

   (b) For each predicate symbol $\mathtt{p}$ in P_E, set

$$\begin{aligned} \mathtt{p}^+ &= \{\langle a_1, \ldots, a_k\rangle \mid \mathtt{p}(a_1, \ldots, a_k) \leftarrow \ \in P_E^\star\}, \text{ and} \\ \mathtt{p}^- &= \{\langle b_1, \ldots, b_k\rangle \mid k \text{ is the arity of } \mathtt{p}, \text{ and } \mathtt{p}(b_1, \ldots, b_k) \leftarrow \ \notin P_E^\star\}. \end{aligned}$$

   (c) For each predicate symbol $\mathtt{p}$ in P_I, set $\mathtt{p}^+ = \emptyset$, and $\mathtt{p}^- = \emptyset$.

2. For each equation of the form $\mathtt{p} = D_{\mathtt{p}}$ in $EQN(P_I)$, compute the expression $D_{\mathtt{p}}$ and set $\mathtt{p}$ to the resulting partial relation.

3. If step 2 involved a change in the value of some $\mathtt{p}$, goto 2.

4. Output the final values of all predicate symbols in P_E and P_I. □

It is instructive to execute Algorithm **COMPUTE** on the database of Example 2.2.4, which we reproduce here.

```
r(a,c)
r(b,b)
s(a,a)
p(X)  ←  r(X,Y), ¬p(Y)
p(Y)  ← s(Y,a)
```

After step 1, the predicate variables have the following values:

$$\begin{aligned}
\mathtt{r}^{+} &= \{\langle a,c\rangle, \langle b,b\rangle\},\\
\mathtt{r}^{-} &= \{\langle a,a\rangle, \langle a,b\rangle, \langle b,a\rangle, \langle b,c\rangle, \langle c,a\rangle, \langle c,b\rangle, \langle c,c\rangle\},\\
\mathtt{s}^{+} &= \{\langle a,a\rangle\},\\
\mathtt{s}^{-} &= \{\langle a,b\rangle, \langle a,c\rangle, \langle b,a\rangle, \langle b,b\rangle, \langle b,c\rangle, \langle c,a\rangle, \langle c,b\rangle, \langle c,c\rangle\},\\
\mathtt{p}^{+} &= \emptyset,\\
\mathtt{p}^{-} &= \emptyset.
\end{aligned}$$

Step 1 can be seen to mimic the production of $T_P^F \uparrow 1$. Each iteration of step 2 uses only equation (4.1), reproduced below.

$$\mathtt{p} = (\dot{\pi}_{\mathtt{\{X\}}}(\mathtt{r(X,Y)} \mathbin{\dot{\bowtie}} \dot{-}\mathtt{p(Y)}))[\mathtt{X}] \;\dot{\cup}\; (\dot{\pi}_{\mathtt{\{Y\}}}(\dot{\sigma}_{\mathtt{Z=a}}(\mathtt{s(Y,Z)})))[\mathtt{Y}] \tag{4.1}$$

By applying the operator definitions introduced earlier, $(\dot{\pi}_{\mathtt{\{X\}}}(\mathtt{r(X,Y)} \mathbin{\dot{\bowtie}} \dot{-}\mathtt{p(Y)}))[\mathtt{X}]$ can be seen to be the partial relation

$$\langle \emptyset, \{\langle c\rangle\}\rangle,$$

and $(\dot{\pi}_{\mathtt{\{Y\}}}(\dot{\sigma}_{\mathtt{Z=a}}(\mathtt{s(Y,Z)})))[\mathtt{Y}]$ the partial relation

$$\langle \{\langle a\rangle\}, \{\langle b\rangle, \langle c\rangle\}\rangle.$$

Their union is thus the partial relation

$$\langle \{\langle a\rangle\}, \{\langle c\rangle\}\rangle$$

assigned by step 2 to the variable p. Further iterations of step 2 do not change the value of p. Step 2 can be seen to mimic an application of the T_P^F function.

Part II

Incompleteness in Relational Databases

CHAPTER 5.

REPRESENTING DEGREES OF EXCLUSIVITY IN DISJUNCTIVE DATABASES

Consider the example of a database which maintains the time it takes (in hours) to travel between two cities by road. Let us assume that there is some uncertainty in the information stored in this database. For example, there might be uncertainty in the amount of time required to travel between any two cities as collected from various sources or there might be uncertainty concerning the source and destination cities itself. Traditional methods of representing uncertainty are broadly classified into two categories - probabilities are associated with tuples to represent the degree to which the tuple is believed to be in the relation, or, at the attribute level, a set of values is used to represent the uncertain value. Disjunctive information has been studied in [41, 17, 78, 58, 18, 72, 48, 82, 24, 66, 51]. This paper addresses the latter method of uncertainty representation. Figure 1 is an instance of such a relation, TRAVEL.

TRAVEL		
Src	Dest	Time
C1	C2	4
C2	C3	{3,4}
{C1,C3}	C4	2

Figure 5.1. An uncertain relation, TRAVEL

Here the uncertainty is represented as a set of values, one or more of which are true, depending on whether the disjunctions are to be interpreted inclusively or exclusively. The second row in the above figure represents the information that the time of travel between cities C1 and C2 is either 3 or 4 hours. It is obvious from the context that this disjunction is to be interpreted exclusively. Similarly, the third row indicates that either the time of travel

between C1 and C4 or C3 and C4 is 2 hours. Such a disjunction is not necessarily exclusive. It may be the case that the time of travel is 2 hours for both pairs. The approach that we adopt allows to explicitly represent whether the disjunction is to be interpreted inclusively or exclusively. We allow conjunctions to also appear in the disjunctions thus representing conjunctions that are *possibly true*. The instance in Figure 1 would be represented as shown in Figure 2 in our model. The data structure we introduce, called an *oa-table* , is shown in Figure 2. An *oa-table* consists of sets of disjunctions. Each disjunction corresponds to an *oa-tuple*. Each disjunct of a disjunction is itself a conjunction of a set of tuples. Informally, an *oa-table* is a conjunction of disjunctions of conjuncts. For any predicate symbol R, an *oa-tuple* is of the form $\eta_1 \vee \eta_2 \vee, \ldots, \vee \eta_m$. Here η_i is a conjunction of tuples $t_{i1} \wedge t_{i2} \wedge, \ldots, \wedge t_{ik_i}$. Thus the disjunction $\eta_1 \vee \eta_2 \vee, \ldots, \vee \eta_m$ is to be viewed as the indefinite statement $(R(t_{11}) \wedge \ldots \wedge R(t_{1k_1})) \vee \ldots \vee (R(t_{m1}) \wedge \ldots \wedge R(t_{mk_m}))$. The disjunctions are interpreted as possibilities in the sense that exactly one of the disjuncts is assumed to be the actual real world truth. We adopt the following convention in the figures of *oa-tables*. Solid double lines are used to separate the schema from the data. Solid lines are used to separate *oa-tuples* and dashed lines are used to separate the disjuncts within an *oa-tuple*. An *oa-table* reduces to an ordinary relation when for each *oa-tuple* $m = 1$ and for each $\eta_i, k_i = 1$. Let w_1, w_2 and w_3 be the three *oa-tuples* in the *oa-table* representation of $TRAVEL$ shown in Figure 2. Thus the instance of the relation $TRAVEL$ represents the formula:

TRAVEL(Src,Dest,Time)
(C1,C2,4)
(C2,C3,3)
(C2,C3,4)
(C1,C4,2)
(C3,C4,2)
(C1,C4,2)
(C3,C4,2)

Figure 5.2. oa-table representation of TRAVEL

$$F1 \wedge F2 \wedge F3 \text{ where,}$$

$$F1 = TRAVEL(C1, C2, 4) \text{ corresponding to } w_1$$

$$F2 = (TRAVEL(C2, C3, 3) \vee TRAVEL(C2, C3, 4)) \wedge (\neg TRAVEL(C2, C3, 3) \vee \neg TRAVEL(C2, C3, 4)) \text{ corresponding to } w_2 \text{ and}$$

$$F3 = TRAVEL(C1, C4, 2) \vee TRAVEL(C3, C4, 2) \text{ corresponding to } w_3$$

The *oa-tuple* w_1 consists of a single definite tuple. w_2 is a disjunction of two tuples and since a set of both tuples is not present as one of the disjuncts, the disjunction is interpreted exclusively and this is represented by the negative clause in $F2$. w_3 is a disjunction of two tuples and since a set of both tuples is present as one of the disjuncts, this represents an inclusive disjunction given by $F3$.

The rest of this paper is organized as follows: Section 2 gives a formal definition of *oa-tables* and its information content and defines a notion of inconsistency in *oa-tables*. Section 3 compares the *oa-table* model to other data models for disjunctive databases. Section 4 defines the operators of the extended relational algebra for *oa-tables*. Section 5 gives an example of query evaluation in the *oa-table* model. Section 6 discusses the semantics of disjunctions in *oa-tables* and compares it with other known techniques for inferring negative information in disjunctive databases. Section 7 concludes the paper and discusses some avenues for future work.

5.1 oa-tables

In this section we present a formal definition of an *oa-table* and its information content and discuss the notions of compactness and consistency in an *oa-table* .

5.1.1 Formal definition

We now formalize the notion of an *oa-table* . A *domain* is a finite set of values. The cartesian product of domains $D_1, D_2, \ldots, D_n$ is denoted by $D_1 \times D_2 \times, \ldots, \times D_n$ and is the

set of all tuples $< a_1, \ldots, a_n >$ such that for any $i \in \{1, \ldots, n\}, a_i \in D_i$. An *oa-table scheme* is an ordered list of attribute names $R =< A_1, \ldots, A_n >$. Associated with each attribute name, A_i, is a domain D_i. Then, T is an *oa-table* over the scheme R where,

$$T \subseteq 2^{2^{D_1 \times D_2 \times, \ldots, \times D_n}}$$

An *oa-table* is a *non-empty set of oa-tuples*. An *oa-tuple* $w = \{\eta_1, \eta_2, \ldots, \eta_m\} \in 2^{2^{D_1 \times D_2 \times, \ldots, \times D_n}}, m \geq 1$ represents the formula $\eta_1 \vee \eta_2 \vee \ldots \vee \eta_m$ where $\eta_i = \{t_{i1}, t_{i2}, \ldots, t_{in}\} \in 2^{D_1 \times D_2 \times, \ldots, \times D_n}, n \geq 0$ represents the formula $t_{i1} \wedge t_{i2} \wedge \ldots \wedge t_{in}$.

5.1.2 Information content of an oa-table

Given a scheme R, we define Γ_R and Σ_R as follows:

$$\Gamma_R = \{T \mid T \text{ is an } \textit{oa-table} \text{ over } R\}$$

and

$$\Sigma_R = \{U \mid U \text{ is a set of relations over } R\}$$

We denote tuples by $t_1, t_2, \ldots, t_k$ and relations are represented by $r_1, r_2, \ldots, r_n$. The symbols $\eta_1, \eta_2, \ldots, \eta_m$ represents sets of tuples and $w_1, w_2, \ldots, w_n$ represent *oa-tuples*. Let $T = \{w_1, w_2, \ldots, w_n\}$ be an *oa-table* . The information content of an *oa-table* is given by a mapping $REP : \Gamma_R \rightarrow \Sigma_R$, which is a composition of two other mappings, $REDUCE$ and M, defined as follows:
$M(T) : \Gamma_R \rightarrow \Sigma_R$ is a mapping such that

$$M(T) = \{\{\eta_1, \eta_2, \ldots, \eta_n\} \mid (\forall i, j)[(1 \leq i, j, \leq n)(\eta_i \in w_i) \wedge (\eta_j \in w_j)] \rightarrow (\neg\exists)\eta'_i \in$$
$$w_i \wedge (\neg\exists)\eta'_j \in w_j \wedge (\eta_i \cap \eta'_j \neq \phi) \wedge (\eta_j \cap \eta'_i \neq \phi) \vee (\exists\eta'_i \in w_i) \wedge (\exists\eta'_j) \wedge (\eta_i \cup \eta_j) \subseteq \eta'_i \vee (\eta_i \cup \eta_j) \subseteq \eta'_j\}$$

Examples of $M(T)$ are shown in figure 5.3. The mapping $REDUCE$ is used to eliminate redundant information from an *oa-table* . The following redundancies may be present in an

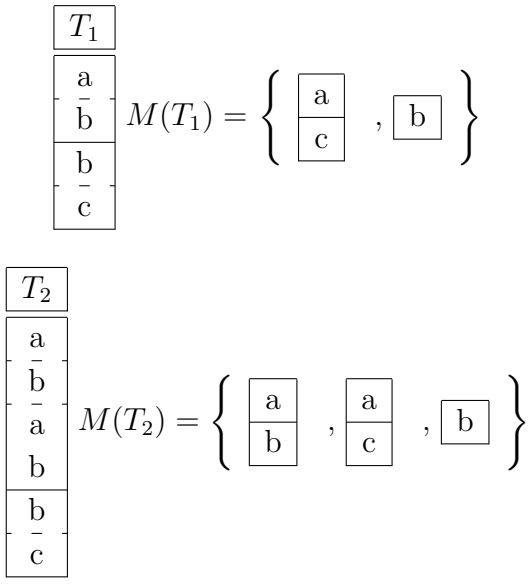

Figure 5.3. $T, M(T)$

oa-table :

1. $w \in T$ and $w' \in T$ and $w \subset w'$ and $\neg\exists\eta' \in w' - w$ and $\eta \subset \eta'$ and $\eta \in w$. This redundancy is eliminated by removing w' from T.

2. $w \in T$ and $w' \in T$ and $w \subset w'$ and $\exists\eta' \in w' - w$ and $\eta \subset \eta'$ and $\eta \in w$. This redundancy is eliminated by removing w from T and setting $w' = w' - \eta''$ where $(\exists\eta \in w)\eta \cap \eta'' = \phi$.

Let T be an *oa-table* . Then, $REDUCE(T) : \Gamma(R) \rightarrow \Gamma(R)$ is a mapping such that $REDUCE(T) = T^0$ where T^0 is defined as follows:

$T^0 = \{w = \{\eta_1, \eta_2, \ldots, \eta_n\} \mid (w \in T) \wedge \neg(\exists w' \in T) \wedge w' \supset w \wedge (\forall\eta' \in w' - w)(\neg\exists\eta \in w)\eta \cap \eta' \neq \phi \wedge \neg(\exists w'' \in T) \wedge w'' \subset w\}$

Figure 6.3 shows an example of REP and $REDUCE$.

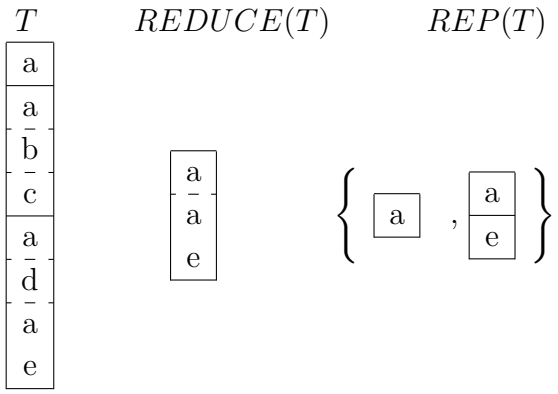

Figure 5.4. $T, REDUCE(T), REP(T)$

5.1.3 Compacting information in oa-tables

We define an operator $COMPACT$, that takes an *oa-table* as input and returns a compact version of the *oa-table*. A set of tuples may appear in every disjunct of a disjunction. Let $w_i = \eta_1 \vee \eta_2 \vee \ldots \vee \eta_m$. Then, $(\forall i)(1 \leq i \leq n \rightarrow \{t_1, t_2, \ldots, t_k\} \subset \eta_i)$. This can be eliminated by the following update:

$$T = T \cup \{\{t_1, t_2, \ldots, t_k\}\}$$

and

$$(\forall i)(1 \leq i \leq m \rightarrow \eta_i = \eta_i - \{t_1, t_2, \ldots, t_k\})$$

$$\begin{aligned} COMPACT(T) = \quad & \{w = \{\eta_1 \vee \eta_2 \vee \ldots \vee \eta_m\} \mid \\ & (\forall i)(1 \leq i \leq m) \rightarrow \\ & \neg\exists\{t_1, t_2, \ldots, t_k\} \mid k \geq 1 \wedge \\ & \{t_1, t_2, \ldots, t_k\} \subset \eta_i\} \end{aligned}$$

5.1.4 Inconsistency in oa-tables

In this section, we define a notion of consistency in *oa tables*. The semantics of disjunctions in an *oa-table* , as defined in section 5.1, introduces inconsistencies. Consider the

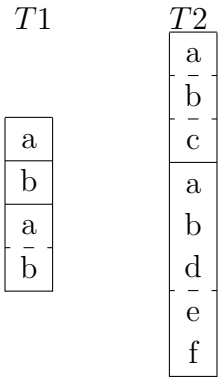

Figure 5.5. Inconsistent oa-tables

oa-tables shown in Figure 5.5. Here, in $T1$, while tuples a and b are in every instance of the relation, the third *oa-tuple* suggests that only one of a or b can be true. This represents inconsistent information. Similarly, in $T2$, the first *oa-tuple* says that exactly one of a, b, c is true. The second *oa-tuple* says that either a, b and d or both e and f are true. $T2$ is inconsistent since the first *oa-tuple* prevents the possibility of a and b both being true. We formally define an inconsistent *oa-table* as follows:

Let T be an *oa-table* and let $\{w, w', w_1, \ldots, w_n\} \subseteq T$. Let $w = \{\eta_1, \eta_2, \ldots, \eta_m\}$ and $s = \cup_i \eta_i$. We denote by $\mathcal{P}(s)$, the powerset of s. Then, T is said to be inconsistent if $\{t_1, t_2, \ldots, t_k\} \subseteq \eta' \in (\mathcal{P}(s) - w) \in w'$ or $w_1 = \{\{t_1\}\}, w_2 = \{\{t_2\}\}, \ldots, w_n = \{\{t_n\}\}$.

Even though inconsistency is possible in *oa-tables*, the main emphasis of this paper is on disjunctions and not on the inconsistency aspect. Therefore, all definitions that follow assume consistent *oa-tables*, unless stated otherwise.

5.2 Related Work

In this section, we explore a few other disjunctive database models and compare them with *oa-tables*.

The *oa-table* model is an extension of the I-tables introduced in [48]. I-tables, while capable of representing indefinite and maybe information, fail to represent certain kinds of incomplete

information. Consider, for example, the following instances of an indefinite database:

```
I1  :  [C1,C2,3]

I2  :  [C1,C2,3]

       [C2,C3,4]

       [C3,C4,1]
```

This set of instances does not have a corresponding I-table representation. The *oa-table* representation for the above instances is shown in Figure 5.6. We restate here the definitions

(C1,C2,3)
(C2,C3,4)
(C3,C4,1)
ϕ

Figure 5.6. oa-table representation of I1,I2

of data models and completeness from [24].

A *data model* (or simply *model*) defines a method for representing an uncertain relation R. A data model M is said to be *complete* if any finite set of relation instances corresponding to a given schema can be modelled by an uncertain relation represented in M.

Theorem 5.2.1. *The oa-table model is complete.*

Proof. Any indefinite database D is a set of instances $\{I_1, I_2, \ldots, I_n\}$ exactly one of which is the real world truth. Since each instance of a database is a set of tuples, the *oa-table* with the single *oa-tuple* $\{I_1, I_2, \ldots, I_n\}$ would represent D. □ □

We also note here that the *oa-table* model is closed under all relational operations which are defined in Section 7.3.

A model M is said to be *closed* if the result of applying an operation on an uncertain relation is also representable in M.

Since the result of applying a relational operation on a finite set of relations is also a finite set of relations and the *oa-table* model is complete, it is also closed.

Another extension of the I-tables is the E-tables discussed by Zhang and Liu in [82]. This model deals with exclusively disjunctive information. Apart from the tuples in $D_1, D_2, \ldots, D_n$, a set of dummy values $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_n$ may also be present as tuples in a relation and they are defined as $\{\varepsilon_i\} = \phi$. Also, $\varepsilon_i \neq \varepsilon_j$ where $i \neq j$. An E-table is a set composed of sets of tuples sets and exactly one element (tuple set) is true in a set of tuple sets making the disjunctions exclusive. This allows the representation of various forms of exclusive disjunctions in E-tables as shown in Figure 5.7. Here, T_1 consists of a single tuple set denoting the fact that either $T_1(a)$ or $T_1(b)$ is true (but not both). E-table T_2 contains dummy values ε_1 and ε_2 which denote dummy 'empty' tuples and this allows the four possible combinations shown in $REP(T_2)$. T_3 contains two tuple sets both of which contain the same dummy value ε. Choosing ε from the first tuple set forces choosing ε from the second one too since the disjunctions in tuple sets are exclusive. This allows only two possible real world scenarios as shown in $REP(T_3)$. Since the E-tables were defined to represent exclusive disjunctions,

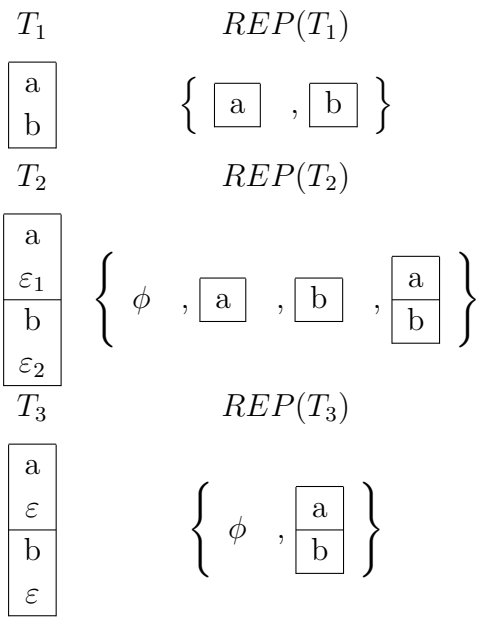

Figure 5.7. E-table representations

the case where the disjunction might be inclusive is not representable in E-tables. i.e., the instance does not have a corresponding E-table representation. *oa-tables*, on the other hand,

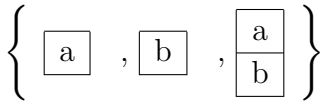

Figure 5.8. An inclusive disjunction

can be used to represent any of the above four degrees of exclusivity as shown in Figure 5.9.

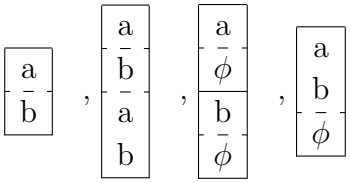

Figure 5.9. Degrees of exclusivity in oa-tables

A third related formalism is the one discussed by Sarma et al. in [24]. That system uses a two-layer approach in which the incomplete model at the top layer has an underlying complete model. Our interest is limited to the complete model described there since the incomplete models are obtained by simply putting restrictions on the complete model. The complete model is a multiset of tuples where or-sets are used to describe uncertain information and existence constraints on the tuples are specified using boolean formulas on the tuples themselves. This approach, although more intuitive than the *c-tables* in [41], still introduces boolean formulas and variables. *oa-tables*, on the other hand, are variable-free and complete.

5.3 Relational Algebra

In this section, we define the operators of the extended relational algebra on Γ_R. We place a dot above the symbol for the operators in order to differentiate them from the standard relational operators.

5.3.1 Selection

Let T' be a consistent *oa-table* and F be a formula involving operands that are constants or attribute names, arithmetic comparison operators:$<, =, >, \leq, \geq, \neq$ and logical operators $\wedge, \vee, \neg$. Then $\dot{\sigma}_F(T') = REDUCE(T)$ where,

$$T = \{w = \{\eta_1, \eta_2, \ldots, \eta_m\} \mid \quad w' \in T' \wedge (\forall i, j)$$
$$((\eta'_i \in w' \wedge t_{ij} \in \eta'_i) \rightarrow$$
$$(\eta_i = \cup_j t_{ij} \mid F(t_{ij})))\}$$

T

a1	b1
a1	b2
a1	b3
a1	b4
a2	b5
a2	b3
a2	b4

$\dot{\sigma}_{1=`a1'}(T)$

a1	b1
a1	b2
a1	b3
a1	b4
ϕ	

$REP(T)$

$\left\{ \right.$

a1	b1
a1	b2
a1	b3
a2	b3

,

a1	b1
a1	b2
a1	b3
a2	b4

,

a1	b1
a1	b4
a2	b5
a2	b3

,

a1	b1
a1	b4
a2	b5
a2	b4

$\left. \right\}$

$REP(\dot{\sigma}_{1=`a1'}(T))$

$\left\{ \right.$

a1	b1
a1	b2
a1	b3

,

a1	b1
a1	b4

$\left. \right\}$

Figure 5.10. Selection

5.3.2 Projection

Projection on *oa-tables* is defined as a mapping $\dot{\pi}_A : \Gamma_R \rightarrow \Gamma_A$ as follows:

Let T' be a consistent *oa-table* and $A \subseteq R$. Then, $\dot{\pi}_A(T') = REDUCE(T)$ where,

$$T = \{w = \{\eta_1, \eta_2, \ldots, \eta_m\} \mid \quad w' \in T' \wedge (\forall i, j)$$
$$((\eta'_i \in w' \wedge t_{ij} \in \eta'_i) \rightarrow$$
$$(\eta_i = \cup_j t_{ij}[A]))\}$$

T

a1	b1
a1	b2
a1	b3
a1	b4
a2	b5
a2	b3
a2	b4

$\dot{\pi}_1(T)$

a1
a1
a1
a2
a2

$REP(T)$

a1	b1	,	a1	b1	,	a1	b1	,	a1	b1
a1	b2		a1	b2		a1	b4		a1	b4
a1	b3		a1	b3		a2	b5		a2	b5
a2	b3		a2	b4		a2	b3		a2	b4

$REP(\dot{\pi}_1(T))$

a1
a2

Figure 5.11. Projection

5.3.3 Cartesian product

Let T_1 and T_2 be consistent *oa-tables* on schemes R_1 and R_2 respectively. Then, the cartesian product of T_1 and T_2, $T_1 \dot{\times} T_2 = REDUCE(T)$ where,

$$T = \{w = \{\eta_{11}, \eta_{12}, \ldots, \eta_{mn}\} \mid \quad (\forall w_1 \in T_1)(\forall w_2 \in T_2)$$
$$((\exists \eta_i \in w_1 \wedge \exists \eta_j \in w_2)$$
$$\rightarrow (\eta_{ij} = (\cup \eta_i \times \cup \eta_j)))\}$$

5.3.4 Union

Let T_1 and T_2 be consistent *oa-tables* on scheme R. Then, $T_1 \dot{\cup} T_2 = REDUCE(T)$ where,

$$T = \{w \mid w \in T_1 \vee w \in T_2\}$$

5.3.5 Intersection

Let T_1 and T_2 be two domain compatible consistent *oa-tables*. Then, $T_1 \dot{\cap} T_2 = REDUCE(T)$ where,

$$\begin{aligned} T = \{w = \{\eta_{11}, \eta_{12}, \ldots, \eta_{mn}\} \mid \quad & (\forall w_1 \in T_1)(\forall w_2 \in T_2) \\ & ((\exists \eta_i \in w_1 \wedge \exists \eta_j \in w_2) \\ & \rightarrow (\eta_{ij} = (\cup \eta_i \cap \cup \eta_j)))\} \end{aligned}$$

5.4 Query Example

In this section we present an example of query evaluation in the *oa-table* model. Consider the database shown in Figure 5.13. This is an instance of a hospital database with two relations `Patient(pname,symptom)` and `Disease(dname,symptom)` which records patient and disease names and their corresponding symptoms. Consider the following query to the database:

`Which patients suffer ONLY from Alzheimer's disease and nothing else?`

The expression for this query in relational algebra is:

$$Q = \dot{\pi}_{<pname>}(\dot{\sigma}_{<2=\text{“}Alzheimer's\text{”}>}(Patient \dot{\bowtie} Disease) - \dot{\sigma}_{<2\neq\text{“}Alzheimer's\text{”}>}(Patient \dot{\bowtie} Disease))$$

The result of evaluating this query is shown in Figure 5.14.

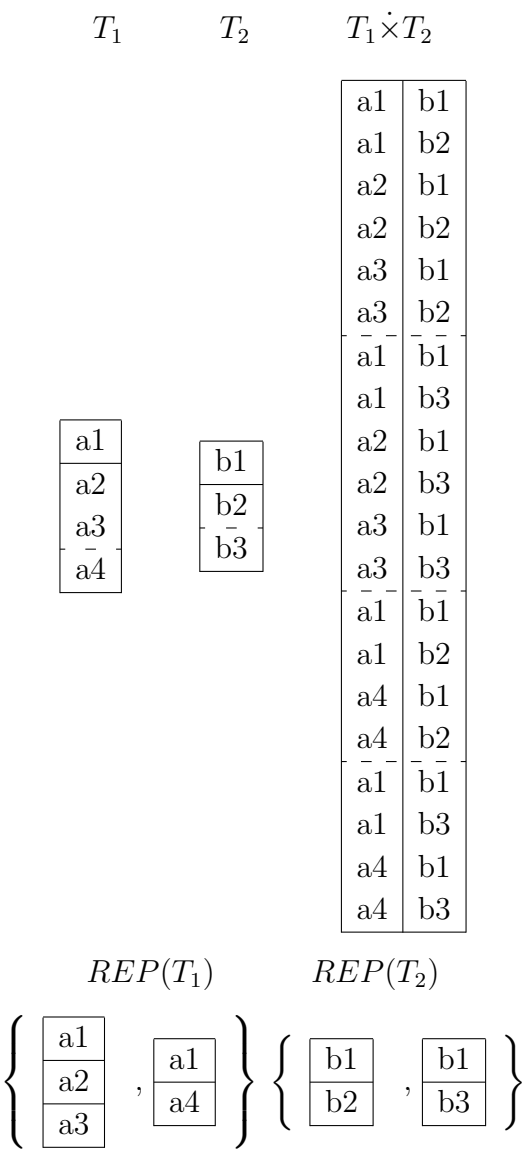

Figure 5.12. Cartesian product

Patient	
pname	symptom
Tom	Forgetfulness
Ann	Forgetfulness
Ann	Fever
Jack	Nausea
Jill	Forgetfulness
ϕ	

Disease	
dname	symptom
Cold	Headache
Cold	Sneezing
Cold	Headache
Alzheimer's	Forgetfulness
Jaundice	Fever
Jaundice	Nausea
Jaundice	Fatigue

Figure 5.13. An instance of a hospital database

$\dot{\sigma}_{<2=\text{“}Alzheimer's\text{''}>}(Patient \dot{\bowtie} Disease)$		
pname	disease	symptom
Tom	Alzheimer's	Forgetfulness
Ann	Alzheimer's	Forgetfulness
Jill	Alzheimer's	Forgetfulness
Ann	Alzheimer's	Forgetfulness
Jill	Alzheimer's	Forgetfulness

$\dot{\sigma}_{<2\neq\text{“}Alzheimer's\text{''}>}(Patient \dot{\bowtie} Disease)$		
pname	disease	symptom
Jack	Jaundice	Nausea
Ann	Jaundice	Fever
Jack	Jaundice	Nausea
ϕ		

Q
pname
Tom
Ann
Jill
Ann
Jill
ϕ

Figure 5.14. Answer to query Q

CHAPTER 6.

D-RELATIONS

Incomplete information is usually present in a database in the form of null values. Several other forms of incompleteness such as fuzzy information, partial values and disjunctive information have been studied extensively. Most of the research on indefinite information has been carried out under the Closed World Assumption(CWA). In this chapter, we present a data structure that supports the opposite view, the Open World Assumption(OWA), where negative information is explicitly represented in a relation. We allow disjunctions at the tuple level to appear in two forms: $A \vee B$ and $\neg A \vee \neg B$, thus obtaining a gain in expressivity. We define a generalization of the relational algebra that handles this kind of information.

Several types of incomplete information have been extensively studied in the past such as *null* values [21, 34, 41, 52], *probabilistic* values [23, 30, 45], *partial* values [53], *fuzzy* and *uncertain* values [12, 61], and *disjunctive* information [35, 47, 48, 49, 75]. In this paper, we present a generalization of the relational model. Our model allows explicit representation of both positive and negative information which may be definite or indefinite. The data structure we introduce, called *d-relations* is a generalization of the paraconsistent relations described in [6] and [7]. A d-relation consists of a positive component and a negative component. The positive component consists of tuple sets where each tuple set represents a positive disjunction and one of the tuples is true in the relation. Similarly, the tuple sets in the negative component represent negative disjunctions and one of the tuples does not belong to the relation.

We extend the ideas of [6] and [48] in d-relations to include definite and indefinite information in both the positive and negative components. There is an interesting interplay between definite and indefinite information. Definite information reduces the uncertainty introduced by disjunctive information: $((P \vee Q \vee R) \wedge \neg P) \equiv (Q \vee R) \wedge \neg P$ and similarly, $(\neg P \vee \neg Q \vee \neg R) \wedge P \equiv (\neg Q \vee \neg R) \wedge P$.

Foundational work on disjunctive databases was done by Minker in [54]. Here, given $P \wedge (P \vee Q)$, we say that P subsumes $P \vee Q$ and we conclude P. The truth value of Q is unknown. This was extended in [48] and [49] to treat Q as *maybe information*. An extension of the relational model where negative information is explicitly represented was presented in [6]. Here, each relation consisted of two components, a positive component containing tuples known to be in the relation and a negative component containing tuples known not to be in the relation. This was extended in [76] to include indefinite information in the form of disjunctions. However, a drawback of this paper was that indefinite information was allowed to appear only in the positive component. The negative component contained only definite information. In this work, we allow negative information - definite and indefinite - to be explicitly stated and show that this increases the expressivity. For instance, if we included the negative clause $\neg P \vee \neg Q$ to the above example, we could infer $P \wedge \neg Q$ from the equivalences above.

6.1 Formal Definition of d-relations

In this section, we formally define the data model called a **d-relation**. We identify several types of inconsistencies and redundancies that may be present in d-relations and present operators to remove them.

Definition 31. *A d-relation R, over a scheme Σ, consists of two components, $< R^+, R^- >$ where $R^+ \subseteq 2^{\tau(\Sigma)}$ and $R^- \subseteq 2^{\tau(\Sigma)}$. R^+, the positive component, is a set of tuple sets. Each tuple set represents a disjunctive positive fact. In the case where the tuple set is singleton, we have a definite positive fact. Similarly, R^-, the negative component, is also a set of tuple sets. Each tuple set in R^- represents a disjunctive negative fact. In the case where the tuple set is singleton, we have a definite negative fact. Let $D(\Sigma)$ denote the set of all d-relations over scheme Σ.* □

Also, we differentiate between the definite and indefinite parts of the positive and negative components. We denote by R_d^+ and R_d^- , the definite positive and negative components of R^+ and R^-, repectively.

$R_d^+ = \{t \mid w \in R^+ \wedge t \in w \wedge |w| = 1\}$

$R_d^- = \{t \mid w \in R^- \wedge t \in w \wedge |w| = 1\}$

We identify conditions under which a d-relation may be inconsistent. A d-relation is said to be inconsistent when every member of a tuple set in the positive(negative) component is present as singleton tuple sets in the negative(positive) component. We deal with this inconsistency by removing the tuple set from the positive(negative) component and the corresponding singleton tuple sets from the negative(positive) component. This is done by the **norm** operator as follows:

Definition 32. *Let R be a d-relation. Then,*

$$\begin{aligned} \mathbf{norm}(\mathbf{R})^+ &= \{w \mid w \in R^+ \wedge w \not\subseteq R_d^-\} - \\ &\quad \{\{t\} \mid (\exists w)(w \in R^- \wedge w \subseteq R_d^+ \wedge t \in w)\} \end{aligned}$$

$$\begin{aligned} \mathbf{norm}(\mathbf{R})^- &= \{w \mid w \in R^- \wedge w \not\subseteq R_d^+\} - \\ &\quad \{\{t\} \mid (\exists w)(w \in R^+ \wedge w \subseteq R_d^- \wedge t \in w)\} \qquad \square \end{aligned}$$

A d-relation is called *normalized* if it does not contain any inconsistencies. Let $\mathcal{N}(\Sigma)$ denote the set of all normalized d-relations on scheme Σ. Fig. 6.1 shows an example of **norm**. We adopt the following convention when representing d-relations: Tuple sets are enclosed within {} and tuples within (). When the d-relation has only one attribute, we drop the () for the tuples. For instance the tuple set $\{(a)\}$ would be written as $\{a\}$. Also, solid lines separate tuple sets and solid double lines separate the positive and negative component.
A normalized d-relation may also contain some redundancies. We identify two types of redundancies that may be present in a d-relation R.

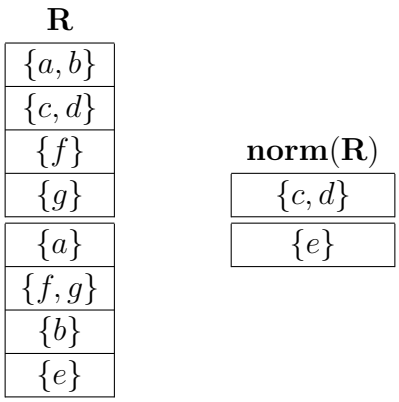

Figure 6.1. An example of **norm**

1. (a) $w \in R^+, w' \in R^+$ and $w \subset w'$. In this case, w subsumes w'. To eliminate this redundancy, remove w' from R^+.

   (b) $w \in R^-, w' \in R^-$ and $w \subset w'$. In this case, w subsumes w'. To eliminate this redundancy, remove w' from R^-.

2. (a) $w \in R^+, v \subseteq R_d^-, v \subset w$. This redundancy is eliminated by removing the tuple set w from R^+ and adding the tuple set $w - v$ to R^+.

   (b) $w \in R^-, v \subseteq R_d^+, v \subset w$. This redundancy is eliminated by removing the tuple set w from R^- and adding the tuple set $w - v$ to R^-.

   Since we are dealing with normalized d-relations, in both cases $w - v$ cannot be empty.

We introduce an operator called **reduce** to take care of redundancies.

Definition 33. *Let R be a normalized d-relation. Then,* $\mathbf{reduce}(\mathbf{R})$ *is defined as follows:*

$\mathbf{reduce}(\mathbf{R}) = \mathbf{subs}(\mathbf{simp}^{\mathbf{lfp}}(\mathbf{R}))$ *where,*

$\mathbf{simp}(\mathbf{R}) =< simp(R)^+, simp(R)^- >$ *where,*

$simp(R)^+ = \{w' \mid (\exists w)(w \in R^+ \wedge w' = w - R_d^-\}$

$simp(R)^- = \{w' \mid (\exists w)(w \in R^- \wedge w' = w - R_d^+\}$

and,

$\mathbf{subs}(\mathbf{R}) =< subs(R)^+, subs(R)^- >$ *where,*

$subs(R)^+ = \{w \mid w \in R^+ \wedge \neg(\exists w_1)(w_1 \in R^+ \wedge w_1 \subset w)\}$

$subs(R)^- = \{w \mid w \in R^- \wedge \neg(\exists w_1)(w_1 \in R^- \wedge w_1 \subset w)\}$

□

The **subs** operator takes care of the first type of redundancy where one tuple set is a subset of another tuple set. The **simp** operator takes care of the second type of redundancy. It takes a d-relation as input and produces another d-relation. $\mathbf{simp^{lfp}}$ in the definition of **reduce** denotes the least fixpoint of the operator **simp**. $\mathbf{simp}(\mathbf{R})$ denotes one application of the operator and $\mathbf{simp^2}(\mathbf{R}) = \mathbf{simp}(\mathbf{simp}(\mathbf{R}))$ and so on. $\mathbf{simp^{lfp}}(\mathbf{R})$ is the d-relation $\mathbf{R}$ satisfying $\mathbf{simp^n}(\mathbf{R}) = \mathbf{R}$ for the least integer $\mathbf{n}$.

Consider the single attribute d-relation $\mathbf{R}$ shown in Fig. 6.2. The tuple set $\{a, b\}$ subsumes

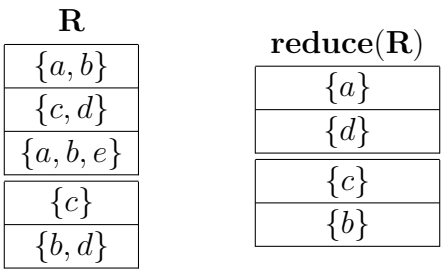

Figure 6.2. An example of **reduce**

$\{a, b, e\}$ in the positive component. Hence the tuple set $\{a, b, e\}$ is deleted. The tuple set $\{c\}$ in the negative component causes c to be removed from the tuple set $\{c, d\}$ in the positive component. Now $\{d\}$ in the positive component causes d to be removed from $\{b, d\}$ in the negative component and the $\{b\}$ in the negative component causes b to be removed from $\{a, b\}$ in the positive component leaving us with the definite d-relation $\mathbf{reduce}(\mathbf{R})$.

A d-relation is a collection of paraconsistent relations. The information content of a d-relation is defined as the set of paraconsistent relations represented by the d-relation. The

different paraconsistent relations represent by a d-relation R can be obtained by choosing one tuple from each tuple set. In doing so, the paraconsistent relations obtained may be inconsistent or may contain redundant information. These would have to be removed from in order to obtain the exact information content of a d-relation.

Definition 34. *Let $U \subseteq \mathcal{P}(\Sigma)$. Then,*

$\mathbf{normrep_{\Sigma}}(\mathbf{U}) = \{R \mid R \in U \wedge R^{+} \cap R^{-} = \varnothing\}$ □

The **normrep** operator removes all inconsistent paraconsistent relations from its input.

Definition 35. *Let $U \subseteq \mathcal{P}(\Sigma)$. Then,*

$$\mathbf{reducerep_{\Sigma}}(\mathbf{U}) = \{R \mid \quad R \in U \wedge \neg(\exists S)(S \in U \wedge R \neq S \wedge S^{+} \subseteq R^{+} \wedge S^{-} \subseteq R^{-})\} \qquad \square$$

The **reducerep** operator keeps only the "minimal" paraconsistent relations and eliminates any paraconsistent relation that is "subsumed" by others.

Definition 36. *The information content of a d-relation is defined by the mapping* $\mathbf{rep_{\Sigma}} : \mathcal{N}(\Sigma) \rightarrow 2^{\mathcal{P}(\Sigma)}$. *Let R be a normalized d-relation on scheme Σ with $R^{+} = \{w_1, w_2, \ldots, w_k\}$ and $R^{-} = \{u_1, u_2, \ldots u_m\}$. Let $U = \{< \{t_1, \ldots, t_k\}, \{s_1, \ldots, s_m\} > \mid (\forall_{i=1}^{k})(\forall_{j=1}^{m})(t_i \in w_i \wedge s_j \in u_j)\}$. Then,*

$\mathbf{rep_{\Sigma}}(R) = \mathbf{reducerep_{\Sigma}}(\mathbf{normrep_{\Sigma}}(\mathbf{U}))$ □

Note that the information content is defined only for normalized d-relations. Fig. 6.3 shows how $\mathbf{rep_{\Sigma}}(\mathbf{R})$ is obtained from a d-relation $\mathbf{R}$. The set of paraconsistent relations denoted by $\mathbf{U}$ in the figure is obtained by the process of selecting tuples from each tuple set. The $\mathbf{normrep_{\Sigma}}$ operator removes the inconsistent paraconsistent relations from $\mathbf{U}$. Finally, the $\mathbf{reducerep_{\Sigma}}$ operator removes the paraconsistent relations that are subsumed by other paraconsistent relations in the set. The following important theorem states that information is neither lost nor gained by removing the redundancies in a d-relation.

Theorem 6.1.1. *Let R be a d-relation on scheme Σ. Then,*

$\mathbf{rep_{\Sigma}}(\mathbf{reduce}(\mathbf{R})) = \mathbf{rep_{\Sigma}}(\mathbf{R})$ □

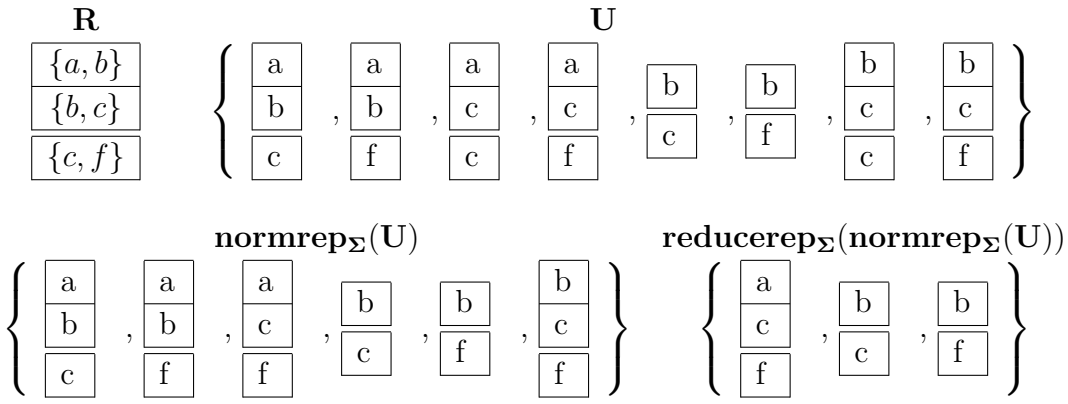

Figure 6.3. An example of $\mathbf{rep_{\Sigma}}$

6.2 Generalized Relational Algebra

In this section, we first develop the notion of *precise generalizations* of algebraic operators. This is an important property that must be satisfied by any new operator defined for d-relations. Then, we present several algebraic operators on d-relations that are precise generalizations of their counterparts on paraconsistent relations.

Precise Generalizations of Operations

It is easily seen that d-relations are a generalisation of paraconsistent relations, in that for each paraconsistent relation there is a d-relation with the same information content, but not *vice versa*. It is thus natural to think of generalising the operations on paraconsistent relations, such as union, join, projection etc., to d-relations. However, any such generalisation should be intuitive with respect to the belief system model of d-relations. We now construct a framework for operators on both kinds of relations and introduce the notion of the precise generalisation relationship among their operators based on [41].

An n-ary *operator on paraconsistent relations with signature* $\langle \Sigma_1, \ldots, \Sigma_{n+1} \rangle$ is a function $\Theta : \mathcal{P}(\Sigma_1) \times \cdots \times \mathcal{P}(\Sigma_n) \to \mathcal{P}(\Sigma_{n+1})$, where $\Sigma_1, \ldots, \Sigma_{n+1}$ are any schemes. Similarly, an n-ary *operator on d-relations with signature* $\langle \Sigma_1, \ldots, \Sigma_{n+1} \rangle$ is a function $\Psi : \mathcal{D}(\Sigma_1) \times \cdots \times \mathcal{D}(\Sigma_n) \to \mathcal{D}(\Sigma_{n+1})$.

We now need to extend operators on paraconsistent relations to sets of paraconsistent relations. For any operator $\Theta : \mathcal{P}(\Sigma_1) \times \cdots \times \mathcal{P}(\Sigma_n) \rightarrow \mathcal{P}(\Sigma_{n+1})$ on paraconsistent relations, we let $\mathcal{S}(\Theta) : 2^{\mathcal{P}(\Sigma_1)} \times \cdots \times 2^{\mathcal{P}(\Sigma_n)} \rightarrow 2^{\mathcal{P}(\Sigma_{n+1})}$ be a map on sets of paraconsistent relations defined as follows. For any sets $M_1, \ldots, M_n$ of paraconsistent relations on schemes $\Sigma_1, \ldots, \Sigma_n$, respectively,

$$\mathcal{S}(\Theta)(M_1, \ldots, M_n) = \{\Theta(R_1, \ldots, R_n) \mid R_i \in M_i, 1 \leq i \leq n\}.$$

In other words, $\mathcal{S}(\Theta)(M_1, \ldots, M_n)$ is the set of Θ-images of all tuples in the cartesian product $M_1 \times \cdots \times M_n$. We are now ready to lead up to the notion of precise operator generalisation.

Definition 37. *An operator Ψ on d-relations with signature $\langle \Sigma_1, \ldots, \Sigma_{n+1} \rangle$ is* consistency preserving *if for any normalized d-relations $R_1, \ldots, R_n$ on schemes $\Sigma_1, \ldots, \Sigma_n$, respectively, $\Psi(R_1, \ldots, R_n)$ is also normalized.* □

Definition 38. *A consistency preserving operator Ψ on d-relations with signature $\langle \Sigma_1, \ldots, \Sigma_{n+1} \rangle$ is a* precise generalisation *of an operator Θ on paraconsistent relations with the same signature, if for any normalized d-relations $R_1, \ldots, R_n$ on schemes $\Sigma_1, \ldots, \Sigma_n$, we have* $\mathbf{rep}_{\Sigma_{n+1}}(\Psi(R_1, \ldots, R_n)) = \mathcal{S}(\Theta)(\mathbf{rep}_{\Sigma_1}(R_1), \ldots, \mathbf{rep}_{\Sigma_n}(R_n))$. □

We now present precise generalisations for the usual relation operators, such as union, join, projection. To reflect generalisation, a hat is placed over an ordinary relation operator to obtain the corresponding d-relation operator. For example, $\bowtie$ denotes the natural join among ordinary relations, $\dot{\bowtie}$ denotes natural join on paraconsistent relations and $\widehat{\bowtie}$ denotes natural join on d-relations.

Definition 39. *Let R and S be two normalized d-relations on scheme Σ. Then, $R \widehat{\cup} S$ is a d-relation over scheme Σ given by $R \widehat{\cup} S = \mathbf{reduce}(T)$, where*

$T^+ = R^+ \cup S^+$ *and*

$T^- = R^- \cap S^-$.

and $R\widehat{\cap}S$ *is a d-relation over scheme* Σ *given by* $R\widehat{\cap}S = \mathbf{reduce}(T)$*, where*

$T^+ = R^+ \cap S^+$ *and*

$T^- = R^- \cup S^-$. □

The positive component of the union is the usual union of the respective positive components of the operands, whereas the negative component of the union is the intersection of the respective negative components of the operands. In the case of intersection the roles of union and intersection are reversed. The intuition behind this and subsequent definitions are derived from the belief system basis for d-relations.

The following theorem establishes the *precise generalization* property for union and intersection:

Theorem 6.2.1. *Let* R *and* S *be two normalized d-relations on scheme* Σ*. Then,*

1. $\mathbf{rep}_\Sigma(R\widehat{\cup}S) = \mathbf{rep}_\Sigma(R)\mathcal{S}(\dot{\cup})\mathbf{rep}_\Sigma(S)$.

2. $\mathbf{rep}_\Sigma(R\widehat{\cap}S) = \mathbf{rep}_\Sigma(R)\mathcal{S}(\dot{\cap})\mathbf{rep}_\Sigma(S)$. □

Definition 40. *Let* R *be a normalized d-relation on scheme* Σ*. Then* $\widehat{-}R$ *is a d-relation over scheme* Σ *given by* $\widehat{-}R^+ = R^-$ *and* $\widehat{-}R^- = R^+$

Definition 41. *Let* R *be a normalized d-relation on scheme* Σ*, and let* F *be any logic formula involving attribute names in* Σ*, constant symbols (denoting values in the attribute domains), equality symbol* $=$*, negation symbol* $\neg$*, and connectives* $\vee$ *and* $\wedge$*. Then, the* selection *of* R *by* F*, denoted* $\widehat{\sigma}_F(R)$*, is a d-relation on scheme* Σ*, given by* $\widehat{\sigma}_F(R) = \mathbf{reduce}(T)$*, where*

$$
\begin{aligned}
T^+ = \quad & \{w \mid w \in R^+ \wedge (\forall t)(t \in w \rightarrow F(t))\} \text{ and} \\
T^- = \quad & \{\{t\} \mid t \in \tau(\Sigma) \wedge \neg F(t)\} \cup \\
& \{w \mid w \in R^- \wedge (\forall t)(t \in w \rightarrow F(t))\}
\end{aligned}
$$

where σ_F *is the usual selection of tuples.* □

A disjunctive tuple set is either selected as a whole or not at all. All the tuples within the tuple set must satisfy the selection criteria for the tuple set to be selected.

Definition 42. *Let R be a normalized d-relation on scheme Σ, and $\Delta \subseteq \Sigma$. Then, the* projection *of R onto Δ, denoted $\widehat{\pi}_{\Delta}(R)$, is a disjunctive paraconsistent relation on scheme Δ, given by $\widehat{\pi}_{\Delta}(R) = \mathbf{reduce}(T)$, where*

$$T^{+} = \{\pi_{\Delta}(w) | w \in R^{+}\} \text{ and}$$

$$T^{-} = \{\{w_1, \ldots, w_k\} \mid \left(\overset{k}{\underset{i=j=1}{\forall}} i, j\right) [(t \in w_i \rightarrow t^{\Sigma} \cap R_d^{+} = \varnothing) \wedge$$
$$(\forall t \in w_i)(\forall t_1, t_2 \in t^{\Sigma})(\exists w, w' \in R^{-})$$
$$(t_1 \in w \wedge t_2 \in w' \wedge w \neq w') \wedge$$
$$(\forall s, t)[(s \in w_i \wedge t \in w_j \wedge i \neq j) \rightarrow$$
$$(\forall s_1, t_1)(\exists w, w' \in R^{-})$$
$$(s_1 \in s^{\Sigma} \wedge t_1 \in t^{\Sigma} \wedge s_1 \in w \wedge t_1 \in w' \wedge$$
$$w \neq w')]]\}$$

where π_{Δ} is the usual projection over Δ of tuples. □

The positive component of the projections consists of the projection of each of the tuple sets onto Δ and $\widehat{\pi}_{\Delta}(R)^{-}$ consists of tuple sets where each extension of each tuple in each tuple set appears in different tuple sets in R^{-}.

Definition 43. *Let R and S be normalized d-relations on schemes Σ and Δ, respectively with $R =< \{u_1, \ldots, u_k\}, \{v_1, \ldots, v_l\} >$ and $S =< \{w_1, \ldots, w_m\}, \{x_1, \ldots, x_n\} >$. Then, the* natural join *of R and S, denoted $R \widehat{\bowtie} S$, is a d-relation on scheme $\Sigma \cup \Delta$, given by $R \widehat{\bowtie} S =$ $\mathbf{reduce}(T)$, where T is defined as follows. Let $E = \{< \{p_1, \ldots, p_k\}, \{q_1, \ldots, q_l\} > | (\forall i)(1 \leq i \leq k \rightarrow p_i \in u_i) \wedge (\forall j)(1 \leq j \leq l \rightarrow q_j \in v_j)\}$ and $F = \{< \{r_1, \ldots, r_m\}, \{s_1, \ldots, s_n\} > | (\forall i)(1 \leq i \leq m \rightarrow r_i \in w_i) \wedge (\forall j)(1 \leq j \leq n \rightarrow s_j \in x_j)\}$ Let the elements of E be $E_1, \ldots, E_e$ and those of F be $F_1, \ldots, F_f$ and let $A_{ij} = E_i \dot{\bowtie} F_j$ for $1 \leq i \leq e$ and $1 \leq j \leq f$. Let $A_1, \ldots, A_g$ be the distinct A_{ij}s. Then,*

$$T^+ = \{v| \quad (\exists s_1)\cdots(\exists s_g)(s_1 \in A_1^+ \wedge \cdots \wedge s_g \in A_g^+ \wedge$$
$$v = \{s_1, \ldots, s_g\})\}$$
$$T^- = \{w| \quad (\exists t_1)\cdots(\exists t_g)(t_1 \in A_1^- \wedge \cdots \wedge t_g \in A_g^- \wedge$$
$$w = \{t_1, \ldots, t_g\})\}$$

□

Theorem 6.2.2. *Let R and S be two normalized disjunctive paraconsistent relations on scheme Σ_1 and Σ_2. Also let F be a selection formula on scheme Σ_1 and $\Delta \subseteq \Sigma_1$. Then,*

1. $\mathbf{rep}_{\Sigma_1}(\widehat{\sigma}_F(R)) = \mathcal{S}(\dot{\sigma}_F)(\mathbf{rep}_{\Sigma_1}(R))$.
2. $\mathbf{rep}_{\Sigma_1}(\widehat{\pi}_\Delta(R)) = \mathcal{S}(\dot{\pi}_\Delta)(\mathbf{rep}_{\Sigma_1}(R))$.
3. $\mathbf{rep}_{\Sigma_1 \cup \Sigma_2}(R \widehat{\bowtie} S) = \mathbf{rep}_{\Sigma_1}(R)\mathcal{S}(\dot{\bowtie})\mathbf{rep}_{\Sigma_2}(S)$. □

Part III

Negation and Nonmonotonic Reasoning

CHAPTER 7.

DEFAULT RELATIONS

In this chapter, we study the issue of incompleteness in an open world database. We define an extension of the relational model which has two forms of negation - the explicit negation, in which certain atoms are known to be false, and a default negation which is a form of non-monotonic negation for unknown atoms in the relation. We define operators for this extended relational model. We show that this model is a generalization of the relational model in the sense that we obtain some intuitive answers in the negative component in addition to the answers obtained in the relational model.

Relational databases normally adopt the CWA. The reason is that the number of negative facts to be stored become prohibitively large and storing them explicitly is not feasible. But this becomes necessary in certain domains of application and when the knowledge is incomplete, a default form of negation must be used. Logical entailment by itself is limited in application when the knowledge is incomplete. But in common sense reasoning, in practice, we do reason about things that we are not completely aware of. A typical example of such a form of reasoning is the statement "*birds fly*".i.e., in general we tend to assume that all birds fly unless we have strong enough reasons to believe otherwise. Consider a particular bird, say Tweety. We would normally assume that Tweety flies as long as we have no reason to believe otherwise. The pattern of reasoning followed here is "*in the absence of information to the contrary ...*". This form of reasoning is nonmonotonic because if we were to subsequently acquire information to the contrary, then we would have to retract our original beliefs. For example, if we were to discover at a later point in time that Tweety is in fact an ostrich, then we would have to retract our earlier belief that Tweety flies. This problem has been studied in detail from the logic programming and deductive database aspect in [33],[36], [37] and [57]. However, this problem has not been studied extensively from the open world relational database viewpoint. In an open world database, some atoms are explicitly defined

and we can make some common sense assumptions about some others. The problem that we address here is to decide about which atoms we can make assumptions within the realm of some of the operators of the relational algebra. In this chapter we define an extension to the relational model and an algebra where two forms of negation are used - an explicit negation and a nonmonotonic form of negation. We show that this model is a generalization of the relational model in the sense that we obtain some intuitive answers in the negative component in addition to the answers obtained in the relational model.

7.1 Default Negation in a Relational Database

In relational databases that adopt the CWA, we store only the true facts and other facts are implicitly assumed to be false. In an open world setting, a relation is a pair $< R^+, R^- >$, where R^+ is the positive component which stores facts that are *known to be true* of the relation R, and R^- is the negative component which stores facts that are *known to be false* in R. Thus, unlike the CWA where we implicitly assume facts not stored to be false, we do not make such an assumption in the open world setting. Facts that we believe to be false are only the ones stored in R^-. Such a model is described in [6],[7]. Bagai and Sunderraman in [6] and [7] define an algebra for their paraconsistent relational data model which has these two components. However, apart from the facts that are known to be false, we also want to be able to make default assumptions about certain facts when the relation is incomplete. The model described in [6] uses the four-valued logic of Belnap [10] and assigns the default truth value of *unknown* to the missing facts in the relation. Some facts may also appear in both R^+ and R^- thus making the relation R inconsistent. Such facts are assigned the fourth truth value *overdetermined*.

Apart from the facts that are known to be true and those that are known to be false in a relation R, we adopt a form of nonmonotonic negation so that some of the unknown facts can be *assumed to be false*. Notice that this is a form of closed world reasoning in an open world setting. It is necessary when the database is incomplete. The form of nonmonotonic

reasoning that we will adopt here is closely related to the one described in [74]. The algebra that we define with two kinds of negation is a generalization of the paraconsistent algebra of [6] and [7]. Apart from the explicit positive and negative components of the answers obtained in that model, we extend it to produce more intuitive negative answers that we conclude to be *false by default*. The basic idea is that we define to be *false by default* certain atoms, as yet unknown, whose addition to the corresponding relation would not have changed the positive consequences of the result of applying a relational operation. It must be noted, however that adding them to the R^- component may change the negative consequences of the relational operation.

7.2 Default Relations

In this section, we construct a set theoretic formulation of our model. In this model, some tuples are *known* to hold a certain underlying predicate, some are *known not* to hold the predicate and some others are *not known* to hold the predicate.

Definition 44. *A default relation on scheme Σ is a triple $< R_e^+, R_e^-, R_d^- >$ where R_e^+,R_e^- and R_d^- are any subsets of $\tau(\Sigma)$. We let $\mathcal{D}(\Sigma)$ denote the set of all default relations on Σ.*

Here, the subscript e denotes explicit and the subscript d denotes default. The superscripts + and - denote true and false respectively. Hence the three components are explicitly true, explicitly false and default false respectively.

Intuitively, R_e^+ may be considered as the set of tuples for which R is known to be true, R_e^- is the set of tuples for which R is known to be false and R_d^- is the set of tuples for which R is not known to be true and hence can be assumed to be false by default.

We denote by $\bar{R}$ the set of tuples on scheme Σ that have not been assigned truth values. Thus $\bar{R} = \tau(\Sigma) - (R_e^+ \cup R_e^- \cup R_d^-)$. We say that a tuple t is *unknown in* R if $t \in \bar{R}$.

Definition 45. *A default relation R on scheme Σ is said to be complete if $R_e^+ \cup R_e^- = \tau(\Sigma)$. R is said to be a consistent default relation if $R_e^+ \cap R_e^- = \emptyset$ and $R_e^+ \cap R_d^- = \emptyset$. If R is both consistent and complete, it is said to be total.*

It should be observed that the (positive parts of) total relations are essentially ordinary relations. We make this relationship explicit by defining an operator $\lambda_\Sigma(R) = R_e^+$ where R is a total relation on Σ.

Default relations are a generalization of ordinary relations in the sense that for each ordinary relation there is a default relation with the same information content. We adopt the notions of generalizations discussed in [6].

An *n-ary* operator on ordinary relations with signature $< \Sigma_1, \ldots, \Sigma_{n+1} >$ is a function $\Theta : \mathcal{O}(\Sigma_1) \times \ldots \times \mathcal{O}(\Sigma_n) \rightarrow \mathcal{O}(\Sigma_{n+1})$ where $\Sigma_1, \ldots, \Sigma_{n+1}$ are any schemes. Similarly, an *n-ary* operator on default relations with signature $< \Sigma_1, \ldots, \Sigma_{n+1} >$ is a function $\Psi : \mathcal{D}(\Sigma_1) \times \ldots \times \mathcal{D}(\Sigma_n) \rightarrow \mathcal{D}(\Sigma_{n+1})$.

Definition 46. *An operator Ψ on default relations with signature $< \Sigma_1, \ldots, \Sigma_{n+1} >$ is totality preserving if for any total relations $R_1, \ldots, R_n$ on schemes $\Sigma_1, \ldots, \Sigma_n$ respectively, $\Psi(R_1, \ldots, R_n)$ is also total.*

We associate with a consistent default relation R the set of all relations obtainable from R by throwing in the missing tuples. The completion of a consistent default relation R is given by,

$$\begin{aligned} comps_\Sigma(R) &= \{Q \in \mathcal{O}(\Sigma) \mid R_e^+ \subseteq Q \subseteq \\ &\qquad \tau(\Sigma) - (R_e^- \cup R_d^-)\} \end{aligned}$$

For any operator $\Theta : \mathcal{O}(\Sigma_1) \times \ldots \times \mathcal{O}(\Sigma_n) \rightarrow \mathcal{O}(\Sigma_{n+1})$ on ordinary relations, we let $\Gamma(\Theta) : 2^{\mathcal{O}(\Sigma_1)} \times \ldots \times 2^{\mathcal{O}(\Sigma_n)} \rightarrow 2^{\mathcal{O}(\Sigma_{n+1})}$ be a map on sets of ordinary relations defined as follows: For any sets $M_1, \ldots, M_n$ of ordinary relations on schemes $\Sigma_1, \ldots, \Sigma_n$ respectively,

$$\begin{aligned} \Gamma(\Theta)(M_1, \ldots, M_n) &= \{\Theta(R_1, \ldots, R_n) \mid R_i \in \\ &\qquad M_i, \forall i, 1 \leq i \leq n\}. \end{aligned}$$

Definition 47. *An operator Ψ on default relations with signature $< \Sigma_1, \ldots, \Sigma_{n+1} >$ is consistency preserving if for any consistent default relations $R_1, \ldots, R_n$ on schemes $\Sigma_1, \ldots, \Sigma_n$ respectively, $\Psi(R_1, \ldots, R_n)$ is also a consistent default relation.*

Definition 48. *A consistency preserving operator Ψ on default relations with signature $< \Sigma_1, \ldots, \Sigma_{n+1} >$ is a strong generalization of an operator Θ on ordinary relations with the same signature, if for any consistent relations $R_1, \ldots, R_n$ on schemes $\Sigma_1, \ldots, \Sigma_n$ respectively, we have*

$comps_{\Sigma_{n+1}}(\Psi(R_1, \ldots, R_n)) = \Gamma(\Theta)(comps_{\Sigma_1}(R_1), \ldots, comps_{\Sigma_n}(R_n))$.

7.3 Algebraic Operators on Default Relations

In this section, we present generalizations of each of the algebraic operators on ordinary relations. To reflect generalization, a dot is placed over the ordinary relational operator to obtain the corresponding default relation operator. The operators defined here are extensions of the operators defined in the paraconsistent data model in [6] and [7]. We also state theorems on strong generalization for each of the operators.

Definition 49. *Let R and S be default relations on scheme Σ. The union of R and S, denoted $R \dot{\cup} S$, is a default relation on scheme Σ, given by,*

$$\begin{aligned} (R \dot{\cup} S)_e^+ &= R_e^+ \cup S_e^+ \\ (R \dot{\cup} S)_e^- &= R_e^- \cap S_e^- \\ (R \dot{\cup} S)_d^- &= R_d^- \cap S_d^- \end{aligned}$$

The union operation may be understood as follows: The tuples in the union of R and S are those that possess either the property R or the property S, which is simply the union of the tuples in R_e^+ and S_e^+ . Similarly, the explicit negation of the union is the tuples which have neither property. They are exactly the tuples in $R_e^- \cap S_e^-$. The tuples *not known to possess* property R or S are those that are not known to possess either - which is exactly the set $R_d^- \cap S_d^-$. Among the unknown tuples in $\bar{R}$ and $\bar{S}$, any of those, if added to either of the original relations, would be present in the union as well. Hence none of them can be negated by default.

Theorem 7.3.1. *The operator $\dot{\cup}$ on default relations is a strong generalization of the operator $\cup$ on ordinary relations.*

Definition 50. *Let R and S be default relations on scheme Σ. The intersection of R and S, denoted $R\dot{\cap}S$, is a default relation on scheme Σ, given by,*

$$\begin{aligned}(R\dot{\cap}S)_e^+ &= R_e^+ \cap S_e^+ \\ (R\dot{\cap}S)_e^- &= R_e^- \cup S_e^- \\ (R\dot{\cap}S)_d^- &= R_d^- \cup S_d^- \cup (\bar{R} \cap \bar{S})\end{aligned}$$

For the intersection operation, the positive component of the intersection will contain exactly those tuples which possess both properties R and S. These are the tuples in $R_e^+ \cap S_e^+$. The tuples in the explicit negative component are those for which it is not the case that they possess properties R and S. i.e. those tuples that either do not possess R or do not possess S. These are the tuples in $R_e^- \cup S_e^-$. The default negative tuples are those tuples that are not known to possess R and S. They include the tuples in $R_d^- \cup S_d^-$. Apart from these, any tuple in $\bar{R}$ which does not appear in S_e^+ will not appear in the intersection even if it were added to R. It will appear in the explicit negative component of the intersection if it was present in S_e^- . Thus we are interested only in tuples that appear in $\bar{R} \cap \bar{S}$. Notice that this holds only if these tuples were to be added separately in R or S. For if any tuple in $\bar{R} \cap \bar{S}$ were to be added to both R and S simultaneously, this tuple would appear in the intersection as well.

Theorem 7.3.2. *The operator $\dot{\cap}$ on default relations is a strong generalization of the operator $\cap$ on ordinary relations.*

Definition 51. *Let R and S be default relations on scheme Σ. The difference of R and S, denoted $R\dot{-}S$, is a default relation on scheme Σ, given by,*

$$\begin{aligned}(R\dot{-}S)_e^+ &= R_e^+ \cap S_e^- \\ (R\dot{-}S)_e^- &= R_e^- \cup S_e^+ \\ (R\dot{-}S)_d^- &= R_d^- \cup (\bar{R} \cap \bar{S}) \cup (\bar{S} - R_e^-)\end{aligned}$$

The tuples in the difference of R and S are those that are in R and not in S. i.e., in $R_e^+ \cap S_e^-$. The tuples that are known not to be present in the difference are exactly those

that are not in R or in S - the tuples in $R_e^- \cup S_e^+$. Any tuple not known to be in R can be assumed to not be in the difference - this is the set R_d^- . Any tuple in $\bar{R} \cap \bar{S}$ would not affect the result of the difference if it were to be added to R or S. Hence they can be assumed to be false by default. Also, the tuples in $\bar{S}$ would not affect the difference even if they were to be added to S. However, some of them already appear in the explicit negative component because they are present in R_e^- . Thus the tuples from $\bar{S}$ that can be assumed to false in the difference are those in $\bar{S} - R_e^-$.

Theorem 7.3.3. *The operator $\dot{-}$ on default relations is a strong generalization of the operator $-$ on ordinary relations.*

If Σ and Δ are relation schemes such that $\Delta \subseteq \Sigma$, then for any tuple $t \in \tau(\Delta)$ we let t^Σ denote the set $\{t' \in \tau(\Sigma) \mid t'(A) = t(A), \text{ for all } A \in \Delta\}$ of all extensions of t. We extend this notion for any $T \subseteq \tau(\Delta)$ by defining $T^\Sigma = \bigcup_{t \in T} t^\Sigma$.

Definition 52. *Let R be a default relation on scheme Σ and let F be any formula involving attribute names in Σ, constant symbols (denoting values in the attribute domains), the equality symbol $=$, the negation symbol $\neg$, and the connectives $\wedge$ and $\vee$. Then, the selection of F by R, denoted $\dot{\sigma}_F(R)$, is a default relation on scheme Σ, given by*

$$\begin{aligned} \dot{\sigma}_F(R)_e^+ &= \sigma_F(R_e^+) \\ \dot{\sigma}_F(R)_e^- &= R_e^- \cup \sigma_{\neg F}(\tau(\Sigma)) \\ \dot{\sigma}_F(R)_d^- &= R_d^- \end{aligned}$$

The positive component of the selection consists of exactly those tuples in R_e^+ that satisfy F. i.e. the tuples that possess property R and satisfy the formula F. The explicitly negated component of a selection includes the set of all tuples in R_e^- since they do not possess property R. Also, tuples in $\tau(\Sigma)$ that do not satisfy F are also explicitly negated. The tuples in R_d^- can be assumed to be false in the selection since they are not known to possess property R.

Theorem 7.3.4. *The operator $\dot{\sigma}$ on default relations is a strong generalization of the operator σ on ordinary relations.*

Definition 53. *Let R be a default relation on scheme Σ, and $\Delta \subseteq \Sigma$. Then, the projection of R onto Δ, denoted $\dot{\pi}_{\Delta}(R)$, is a default relation on Δ, given by*

$$\begin{aligned}
\dot{\pi}_{\Delta}(R)_e^+ &= \pi_{\Delta}(R_e^+) \\
\dot{\pi}_{\Delta}(R)_e^- &= \{t \in \tau(\Delta) \mid t^{\Sigma} \subseteq R_e^-\} \\
\dot{\pi}_{\Delta}(R)_d^- &= \{t \in \tau(\Delta) \mid t^{\Sigma} \subseteq (R_d^- \cup R_e^-)\} \\
&\quad -\dot{\pi}_{\Delta}(R)_e^-
\end{aligned}$$

The positive component of the projection is the projection of tuples in R_e^+ . The explicitly negated component of the projection is those tuples in Δ all of whose extensions are explicitly negated in R. Similarly, the tuples that are unknown in Δ all of whose extensions are in R_d^- can be assumed to be false in the projection. Apart from this, there may be tuples unknown in Δ some of whose extensions are in R_e^- and the others in R_d^- . These tuples can also be assumed to be false by default.

Theorem 7.3.5. *The operator $\dot{\pi}$ on default relations is a strong generalization of the operator π on ordinary relations.*

Definition 54. *Let R and S be default relations on scheme Σ and Δ respectively. Then, the natural join of R and S, denoted $R\dot{\bowtie}S$, is a default relation on scheme $\Sigma \cup \Delta$, given by*

$$\begin{aligned}
(R\dot{\bowtie}S)_e^+ &= R_e^+ \bowtie S_e^+ \\
(R\dot{\bowtie}S)_e^- &= (R_e^-)^{\Sigma\cup\Delta} \cup (S_e^-)^{\Sigma\cup\Delta} \\
(R\dot{\bowtie}S)_d^- &= (R_d^-)^{\Sigma\cup\Delta} \cup (S_d^-)^{\Sigma\cup\Delta} \cup \\
&\quad \{t^{\Sigma\cup\Delta} \mid (t \in \bar{R} \wedge \{t\} \bowtie S_e^+ = \emptyset) \vee \\
&\qquad (t \in \bar{S} \wedge R_e^+ \bowtie \{t\} = \emptyset)\}
\end{aligned}$$

The positive component of the join is simply the natural join of the positive components of the corresponding relations. The explicitly negated component of the join consists of all extensions of the tuples in R_e^- and S_e^- since these are already not true in R and S respectively. The default negative component consists of all extensions of R_d^- and S_d^- . This component

will also contain extensions of tuples in $\bar{R}$ that do not join with any tuple in S_e^+ . Similarly, we can also add all tuples from $\bar{S}$ that do not join with any tuple in R_e^+ .

Theorem 7.3.6. *The operator $\dot{\bowtie}$ on default relations is a strong generalization of the operator $\bowtie$ on ordinary relations.*

7.4 Intuition

The aim of this section is to explain the intuition behind how the default conclusions are made for each of the relational operators. Since the semantics of each of the relational operators is clear, we know what kind of inferences can be made at least as far as the positive conclusions are concerned. For example, we know that the union of two relations is exactly those set of tuples that are known to have either property. In order to derive default conclusions, we are motivated by two reasons - one, we want to minimize the extent of a relation. This is the idea behind nonmonotonic reasoning methods like circumscription [53]. Thus we attempt to derive default negative conclusions in order to minimize the result of an algebraic operation. The question that then arises is on what basis do we minimize? As mentioned earlier, since the semantics of the relational operators are clear, an interesting approach would be to treat this as the definite result of applying the particular relational operator and try to minimize the relation as much as possible while maintaining consistency without introducing any change in the definite answers.This is the second motivation for deriving default conclusions. This approach leads to the obvious question - why not negate every unknown fact as in the CWA? The reason that this approach is unsatisfactory is that we want to differentiate between two kinds of negation in an open world database. The explicit negation component is the set of tuples whose falsity has been constructively established. The default negation component is the set of tuples whose falsity can be assumed. The need for distinguishing between these two forms of negation has been studied extensively from the logic programming perspective. In particular, PROLOG's negation operator *not* is a nonmonotonic form of negation based on the negation as failure rule due to Clark [20].

Most default assumptions are made on the inference "in the absence of any information to the contrary, assume ...". Our attempt here is to find a formalization of this principle in relational databases. For the relational operators, since our effort is to minimize the resulting relation, we assume that a tuple is not in the result of a relational operation unless we have good enough reasons to believe otherwise. Since both the explicit positive and negative components of the result of a relational operation are defined as functions of the corresponding components of the input relations, for each tuple that is unknown in the input relations, we assume the tuple to be in the relation and then compute the result. If there is no change in the result, then we conclude that the tuple can be negated by default in the result. For the purpose of illustration, consider a default relation R and the selection of R by a formula F. The positive component of the selection is defined in terms of R_e^+ . Among the tuples that are unknown in R, there are some for which F holds and others for which it does not. Consider the tuples for which F does not hold. Even if they were to be in R, the result of selection would be the same since F does not hold for them. Hence these are the only tuples in $\bar{R}$ that can be negated by default. Notice here that this form of negation is stronger than the CWA since the CWA dictates that all tuples for which R does not hold are assumed to be false. Since we are dealing with the open world assumption we need a stronger notion of negation.

CHAPTER 8.

EXTENDED LOGIC PROGRAMS

The problem of assigning a semantics to logic programs with negation(called normal logic programs in this chapter) has been heavily studied. Of the various semantics proposed, two of the most popular semantics are the wellfounded semantics [81] and the stable model semantics [32]. For programs not containing any negation, the *Horn programs*, there is a consensus on the semantics. Even for restricted classes of logic programs with negation, the *stratified and locally stratified programs*, there is an agreement on the semantics. Thus it may be said that the various semantics for logic programs with negation differ primarily in their treatment of the negation. Extended logic programs are logic programs with two types of negation; the *weak negation*, of the kind seen in a normal logic program, and another kind of negation, which we will call *explicit negation* in this chapter. The explicit negation may be thought of as a rough counterpart of the classical negation of first order logic.

The weak negation of normal logic programs (not containing explicit negation) is similar in spirit to the Closed World Assumption(CWA) of Reiter [64]. The CWA assumes the negation of a statement when the statement cannot be proved. Hence there is no need to state negative information explicitly. It can be seen that in extended logic programs we are no longer in a closed world setting. However, most semantics for extended logic programs treat the weak negation in an extended logic program in the same way that it is treated in a normal logic program. We argue that when normal logic program semantics are used to specify the semantics of an extended logic program, the weak negation should be treated differently. Surely, when we are trying to establish the "failure to prove" a proposition, we must pay attention to "proving the negation of the proposition". When a user specifies an extended logic program containing weak negation, the weak negation of a literal may either mean that the literal is false (which may be stated through explicit negation) or that we may apply the CWA for the literal. The aim of this chapter is to attempt to formalize the

meaning of the weak negation through a translation to normal logic programs so that normal logic program semantics may be applied to the extended logic program.

8.1 Preliminary Definitions and Motivation

We first recall basic definitions for logic programs. A *term* is a constant, a variable or a complex term of the form $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and f is a *function symbol* with finite arity $n \geq 0$. An *atom* is a formula of the language of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of finite arity $n \geq 0$ and $t_1, \ldots, t_n$ are terms. A *literal* is either an atom or its negation, denoted by $p(t_1, \ldots, t_n)$. An atom A and its negation $\neg A$ are said to be *complements* of each other. In general, if B is a literal, $\neg B$ denotes the complement of B.

A normal logic program is a set of rules of the form

$$A \leftarrow B_1, \ldots B_n, \sim C_1, \ldots, \sim C_m \tag{8.1}$$

where $A, B_1, \ldots B_n, C_1, \ldots C_m$ are atoms and $\sim$ denotes weak negation. Here A is called the head of the rule and the conjunction $B_1 \wedge \ldots \wedge B_n \wedge C_1 \wedge \ldots \wedge C_m$ is called the body of the rule. When $m = 0$, we call the rule a "strict" rule, and a "defeasible" rule otherwise.

An extended logic program is a set of rules of the form (8.1) but here $A, B_1, \ldots B_n$, $C_1, \ldots, C_m$ are literals, i.e., they are of the form P or $\neg P$. An extended logic program thus contains two types of negation - the "weak negation" $\sim$ of the kind seen in a normal logic program and an "explicit negation" $\neg$.

Given a logic program P, the *Herbrand universe* of P, denoted U_P, is the set of all possible ground terms constructed recursively using the constants and function symbols occuring in P. The *Herbrand base* of P, denoted HB_P, is the set of all possible ground atoms whose predicate symbols occur in P and whose arguments are elements of U_P. A term, atom, literal, rule or program is ground if it is free of variables. A *ground instance* of a rule is

obtained by replacing the variables in a program with elements from U_P in every possible way. A ground program is the union of the ground instances of the rules in the program. In this chapter, we assume programs to be ground unless stated otherwise.

Motivation. Consider the normal logic program $a \leftarrow\sim b$. The atom a is a consequence in this program under any semantics. The weak negation in this rule is interpreted as *negation as failure to prove* [20]. This rule essentially says: "conclude a if it is not possible to prove b from the program". Since there is no rule in the program with b in the head, we can never derive b from this program and hence we can conclude a to be true. The negation as failure rule is very close in spirit to the Closed World Assumption (CWA) used in database systems [64]. In a database system, the facts stored in the database are true and the other facts are *assumed to be false*. We assume that our knowledge about the world is *complete*. Thus there is no need to explicitly state that a fact is false. This appears to be appropriate : the number of false facts usually tends to be very large and it is not feasible to store these facts. It is easy to see that extended logic programs do not adopt this view. In an extended logic program, a (classically) negated literal may appear in the head of a rule and we may thus derive a negative fact from the rules of the program. Since negative information is stated explicitly, it is possible for information to be neither true nor false. Such facts are *unknown*. This is known as the Open World Assumption(OWA). Here we admit that our knowledge of the world is *incomplete*. As a result, it is possible for an extended logic program to be contradictory i.e. it is possible to derive both a literal and its negation from the program. Consider for example the following program:

$$a \leftarrow .$$

$$\neg a \leftarrow .$$

It is easy to see that this program is contradictory. The body of the rules in an extended program may also contain $\sim$. Most of the semantics proposed for extended logic programs treat $\sim$ in the same way as it is treated in a normal logic program. Consider the following

program in which the first four rules are from [33] and the last two rules are added for illustrative purposes.

$$
\begin{aligned}
Eligible(X) &\leftarrow HighGPA(X).\\
Eligible(X) &\leftarrow Minority(X), FairGPA(X).\\
\neg Eligible(X) &\leftarrow \neg FairGPA(X).\\
Interview(X) &\leftarrow \sim Eligible(X), \sim \neg Eligible(X).\\
Scholarship(X) &\leftarrow \sim \neg Eligible(X).\\
\neg Scholarship(X) &\leftarrow \sim Eligible(X).
\end{aligned}
$$

These are a set of rules used by a college to determine whether a student should be given a scholarship or not. The first three rules are used to determine eligibility and the fourth rule says that the student should be interviewed in the event that eligibility cannot be determined using the first three rules. Assume that we have the following facts available to us about two students Tom and Ann: $\{FairGPA(Ann), \neg FairGPA(Tom)\}$. We can conclude from this program that Tom is not eligible for a scholarship since he does not have a fair GPA. However, it is not possible to decide eligibility for Ann since she does not have a high GPA and we do not know whether she belongs to a minority or not. We use the fourth rule to conclude that Ann should be interviewed. But now we may use the last two rules to conclude both $Scholarship(Ann)$ and $\neg Scholarship(Ann)$ and the program with the two facts now becomes contradictory.

It appears that the conclusions $Scholarship(Ann)$ and $\neg Scholarship(Ann)$ should have been blocked by making the last two rules inapplicable. Without these two conclusions, the program appears to have a very reasonable semantics in which Tom is found ineligible for the scholarship and Ann is interviewed. It should be noted here that the complementary pair of literals are derived from defeasible rules in the program. This problem arises due to the treatment of $\sim$ as the weak negation in a (closed world) normal logic program. In

an open world setting, we adopt a three-valued logic in which atoms can be true, false or unknown. For an atom p, when p holds the atom is true, it is false when $\neg p$ holds, and it is unknown otherwise. In the closed world setting, since there is no notion of $\neg p$, we assume that p is false when we cannot derive p. For the above program, it is clear that the truth value of the atom $Eligible(Ann)$ (and $\neg Eligible(Ann)$) is unknown because we have not been able to establish $Eligible(Ann)$ or $\neg Eligible(Ann)$. So essentially, we have derived contradictory information using unknown values. If on the other hand, we were able to establish $\neg Eligible(Ann)$ through some other rule then we might have been able to use the sixth rule to determine that Ann should not be given the scholarship (the atom $Eligible(Ann)$ is then false and weak negation may be applied). This is the principle of coherence formulated in [60].

Consider a much simpler program P_0

$$\begin{aligned} \neg a &\leftarrow . \\ a &\leftarrow \sim b. \end{aligned}$$

This program has a contradictory semantics but it is clear that in the open world setting the truth value of the atom b is unknown (since we have both $\sim b$ and $\sim \neg b$). We have thus derived contradictory information from an unknown atom by applying closed world reasoning in an open world setting. On the other hand, if the above program were to consist only of the second rule, we should derive a from the program since then it is still a normal logic program. But when we add the fact $\neg a$, we are now in an open world setting. In that scenario, it does not appear reasonable to use the failure to prove b(when essentially we have that b is unknown since we have failed to prove $\neg b$ as well) to derive the atom a when we have explicit information that a is false.

8.2 Related Work

A large amount of research has gone into semantics for extended logic programs [33, 60, 44, 5, 42]. One of the earliest works in this area was the extension of the stable model semantics for normal logic programs to extended logic programs by Gelfond and Lifschitz in [33]. The stable model semantics was first introduced in [32]. We briefly describe here the answer set semantics for extended logic programs.

Answer set semantics. Let Π be an extended program without variables that does not contain $\sim$, and let Lit be the set of ground literals in the language of Π. The *answer set* of Π is the smallest subset S of Lit such that

1. for any rule $A \leftarrow B_1, \ldots B_n$ from Π, if $B_1, \ldots B_n \in S$, then $A \in S$;

2. if S contains a pair of complementary literals, then $S = Lit$.

Now let Π be *any* extended program. For any set $S \in Lit$, let Π^S be the extended program obtained from Π by deleting

1. each rule that has a formula $\sim L$ in its body with $L \in S$, and

2. all formulas of the form $\sim L$ from the bodies of the remaining rules.

Clearly, Π^S does not contain $\sim$, so that its answer set is already defined. If this answer set coincides with S, then we say that S is an *answer set* of Π. An extended program is considered *contradictory* if it has an inconsistent answer set (an answer set containing a pair of complementary literals).

It can be seen that the answer set semantics amounts to replacing every occurence of a classically negated literal in the program with its primed version(every $\neg p$ is replaced with p') and then applying the stable model semantics to the normal logic program thus obtained. The idea of the transformation of the program by replacing occurences of $\neg p$ with p' appears to be inadequate. Once such a transformation is made, we have essentially lost the connection between a literal and its complement. Also, since we now have a normal

logic program, the weak negation $\sim$ operates in the same way as it does in a normal logic program. We argue that in an open world setting, the meaning of $\sim$ should be "stronger" in some sense. In what sense will be made explicit by the transformation that we will define later.

Another related work is [55]. Here the authors transform an extended (disjunctive) logic program into a normal logic program by a transformation somewhat similar to the one defined by Gelfond and Lifschitz in [33]. The transformation, called a *prime-$\perp$-transformation*, transforms every negated literal $\neg p$ to p' and adds rules of the form $\perp \leftarrow p, p'$ for every literal p in the Herbrand base of the program. The new symbol $\perp$ denotes inconsistency and is not present anywhere in the program. The purpose of the new rules is to to be able to "detect" inconsistency in the program. Following the *prime-$\perp$-transformation*, any semantics may be applied to the normal logic program and the transformations are reversed to obtain the semantics of the extended logic program. However, since the transformation is similar to the one in [33], the programs found to be inconsistent are the same as the ones in the answer set semantics.

It is known that the wellfounded semantics of normal logic programs [81] is an approximation of the stable model semantics. The wellfounded model of a normal logic program may be defined in terms of the Gelfond-Lifschitz operator, henceforth called Γ [32]. Since Γ is anti-monotonic, Γ^2 is monotonic and the wellfounded model may be obtained by iterating Γ^2 from below until it reaches its least fixpoint. Brewka in [14] defines the wellfounded semantics of an extended logic program in terms of this operator with a slight modification. Applying Γ on a set of (extended) literals may result in inconsistency. If the requirement of logical closedness is removed, then we may avoid inconsistency due to the presence of complementary literals. Thus Brewka defines the wellfounded model of an extended logic program as the least fixpoint of $\Gamma\Gamma'$ iterated from below where Γ' is Γ without the requirement of logical closedness.

The work of Alferes and Perreira in [60] defines a well-founded semantics for extended logic programs. Here the authors introduce the so-called *coherence requirement*. The coherence requirement essentially states that if an atom p is false($\neg p$ is a consequence of the program), then $\sim p$ must hold as well. This appears to be reasonable: if a literal is explicitly false, then surely it should be false by default as well. Although the coherence requirement does relate the two forms of negation in an extended logic program, the weak negation $\sim$ still holds the same meaning under their semantics. The work in [2] studies extended logic programs in an abductive framework and proposes a series of semantics that vary in their degree of skepticism.

One of the works in extended logic programs aimed at avoiding inconsistencies is [44]. Here rules with negative literals in the head are referred to as "exceptions" i.e. a literal p may be derived from a rule in an extended logic program unless an exception to it has been generated through the derivation of $\neg p$. The authors propose a simple reformulation of the stable model semantics to handle exceptions.

Arieli in [5] proposes a fixpoint semantics for extended logic programs. He defines an operator that transforms an extended logic program into a normal logic program. Since this work is in the paraconsistent setting, the definition of $\sim$ is given in a four-valued logic. The possible truth values are $\{t, f, \bot, \top\}$ and $\sim t = f$, $\sim f = t$, $\sim \top = f$ and $\sim \bot = \bot$. The semantics proposed handles inconsistent information.

8.3 Program Transformation

In this section, we show how an extended logic program should be transformed to a normal logic program. We argue that the transformations such as the ones defined in [33] and [55] do not reflect the intended meaning of the program in the open world setting.

It has been shown in [33] that the CWA for a predicate can be captured in an extended logic program. For instance, in order to state that a predicate p adopts the CWA, we use the following rule:

$$\neg p \leftarrow\sim p$$

This rule states that we can derive $\neg p$ if we fail to prove p. We use this in order to define transformations for rules containing $\sim$.

For a rule r in an extended logic program P, we denote by $T(r)$, the transformed version of the rule r. Then, the transformation for P is given by

$$T(P) = \bigcup_{r \in P} T(r)$$

We call the *prime transformation* of r, denoted $prime(r)$, the process of replacing every negative literal $\neg A$ in r by its primed version A'.

Consider an arbitrary rule r of the form

$$A \leftarrow B_1, \ldots B_n, \sim C_1, \ldots, \sim C_m$$

There are three separate cases to consider for the transformation of r.

Case 1: When $(m = 0)$ or $(m = 2$ and $C_1 = \neg C_2)$.

$T(r) = prime(r)$.

When $m = 0$, since there is no use of $\sim$, there is only one interpretation of the rule regardless of whether we are operating under the CWA or the OWA. When $m = 2$ and $C_1 = \neg C_2$, the intention of the programmer is to state that C_1 is *unknown* under the OWA i.e. neither C_1 nor $\neg C_1$ can be established. Such rules are treated as rules in which the programmer had the OWA in mind and hence we only perform the prime transformation for these rules.

Case 2: When $m \geq 1$ and $\{C_1, \ldots, C_m\}$ does not contain a pair of complementary literals.

$T(r)$ is given by a pair of rules $prime(r_1)$ and $prime(r_2)$ where

$$\begin{aligned} r1 : A &\leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \neg C_m \\ r2 : A &\leftarrow B_1, \ldots, B_n, \sim C_1, \ldots, \sim C_m, \sim \neg A \end{aligned}$$

Case 3: When $m > 2$ and $\{C_1, \ldots, C_m\}$ contains atleast a pair of complementary literals, say C_i, i.e. $\{C_i, \neg C_i\} \subset \{C_1, \ldots, C_m\}$.

$T(r)$ is given by a pair of rules $prime(r_1')$ and $prime(r_2)$ where

$$r_1' : A \leftarrow B_1, \ldots, B_n, \neg C_1, \ldots, \sim C_i, \sim \neg C_i, \ldots, \neg C_m$$

and $r2$ is the same as in Case 2.

It is easy to see that Case 3 is simply a special case of 2 when there is a pair of complementary literals in the body of the rule. The idea is that in such a case, that part of the body does not require any transformation other than the prime transformation(already explained in Case 1). We give here a brief justification for Case 2.

The intuition behind $r1$(and $r1'$) is that if $C_1, \ldots, C_m$ are to be assumed false by weak negation, then in the open world setting it must be the case that they are false by explicit negation. This the coherence requirement of [60]. Thus A may be derived from the original rule if $B_1 \wedge \ldots \wedge B_n \wedge \neg C_1 \wedge \ldots \wedge \neg C_m$ holds.

Simply using this transformation alone appears to be too strong in the sense that for simple programs such as $\neg a \leftarrow\sim b$ we fail to derive $\neg a$ because we cannot prove b to be false($\neg b$ cannot be proved). Thus $r1$ alone is what might be called a "strictly open world" transformation. With such a transformation, any program gets reduced into a simple Horn program and the meaning of the weak negation $\sim$ is simply the same as the explicit negation

$\neg$. We want to be able to identify a "middle ground" so that $\neg a$, for instance, in the above program is derived. Looking at the rule, it is clear that the programmer intended to say "derive $\neg a$ if we fail to prove b". But since we are operating under the OWA, what the programmer intended to say should be taken as "failure to prove b may be used to derive $\neg a$ provided we use closed world reasoning for a". This assumption can be included in the rule by adding $\sim a$ to the body of the rule. This is achieved by $r2$ in the transformation.Essentially, we are saying that when an extended logic program is specified, if it is to be evaluated under any of the semantics for normal logic programs, the intended meaning of the program under the CWA is given by the transformation above.

This simple transformation enjoys a number of desirable properties.

- For definite logic programs, which are programs with rules of the form $A \leftarrow B_1, \ldots, B_n$ where $A, B_1, \ldots, B_n$ are atoms, the transformation has no effect and we have the same program.

- For normal logic programs too, the transformation has no effect on the semantics of the program, i.e., Then $SEM(P) = SEM(T(P))$, where SEM is any semantics for normal logic programs and $SEM(P)$ is the set of literals entailed by program P under the semantics. In other words, the transformed program is semantically equivalent to the original normal logic program. This is easily observed. First of all, we consider only Case 2 since there is no occurence of complementary literals in a normal logic program. The first transformed rule $r1$ contains classically negated literals in the body. These literals have no rules with them in the head since the original program is a normal logic program. Hence the bodies of these rules will never be satisfied and these rules may be deleted. The second rule $r2$ contains $\sim \neg A$ in its body. Again since there are no rules with $\neg A$ in the head, we may delete it from the bodies of the rules. We are then left with the original normal logic program.

- The transformation results in programs that are not contradictory by most of the semantics for normal logic programs.
- If an extended logic program P is not contradictory by a semantics, say SEM, then $T(P)$ is not contradictory by SEM either.

We illustrate this with a few examples.

Example 8.3.1

Consider the extended logic program

$$a \leftarrow\sim b$$

$$\neg a \leftarrow\sim b$$

The transformed program is

$$\begin{aligned} a &\leftarrow b' \\ a &\leftarrow \sim b, \sim a' \\ a' &\leftarrow b' \\ a' &\leftarrow \sim b, \sim a \end{aligned}$$

The original program is contradictory under the answer set semantics. However, the transformed program has two stable models $\{a\}$ and $\{a'\}$.

Example 8.3.2

Consider the extended logic program

$$\begin{aligned} \neg a &\leftarrow \\ a &\leftarrow \sim b \end{aligned}$$

The transformed program is

$$\begin{aligned} a' &\leftarrow \\ a &\leftarrow b' \\ a &\leftarrow \sim b, \sim a' \end{aligned}$$

This program is again contradictory under the answer set semantics but the only stable model of the transformed program is $\{a'\}$. Notice however that if we were to add the fact $\neg b$ to the original program, then the program would indeed be contradictory under the answer set semantics. This is as expected: although we need to negation as failure of b in order to derive a, since we have $\neg b$, it appears reasonable that a is a consequence in this program.

Example 8.3.3

Consider again the simple program

$$a \leftarrow \sim b$$

This program is written in the closed world setting since $\neg$ does not occur in the program. Consider the case where the programmer intended to specify the same rule in the open world setting, i.e. the programmer wants to state that a is true if there is no evidence of b. Clearly, this would have been done differently. In the view of these authors, this could have been stated either as $a \leftarrow \neg b$, where failure to prove is stated in terms of proving the complementary literal, or as $a \leftarrow\sim b, \sim \neg b$, where $\sim b \wedge \sim \neg b$ indicates no evidence of b in the open world setting. Notice that the latter rule falls under Case 1 in the transformation. The transformation has no effect so that the meaning of "no evidence of b" is preserved. Thus "a is true if there is no evidence of b" should have been stated either as $a \leftarrow \neg b$ or as $a \leftarrow\sim b, \sim \neg b$ in the open world setting.

Given such a rule in the open world setting, it is really not clear whether the programmer intended the negation of b, which should have been stated as $\neg b$, or, the failure to prove b, as in the CWA. The transformation we define tries to account for both cases. For our transformation, this program would have been translated to the two rules $a \leftarrow \neg b$ and $a \leftarrow\sim b, \sim \neg a$. The latter transformation reflects our intuition that failure to prove b may be used to derive a if a follows by adopting the closed world assumption for a. This is specified in the open world setting by adding $\sim \neg a$ to the body of the rule.

Example 8.3.4

Consider the extended logic program

$$\begin{aligned} a &\leftarrow \\ c &\leftarrow a \\ \neg b &\leftarrow \\ \neg c &\leftarrow \sim b \end{aligned}$$

The transformed program is

$$\begin{aligned} a &\leftarrow \\ c &\leftarrow a \\ b' &\leftarrow \\ c' &\leftarrow b' \\ c' &\leftarrow \sim b, \sim c \end{aligned}$$

This program is contradictory by any semantics. Since $\neg b$ is a fact, by the coherence requirement we have $\sim b$ and this can be used to derive $\neg c$, thus deriving both c and $\neg c$. Though $\neg c$ can only be derived through a defeasible rule in the original program, since $\neg b$ can be derived, the rule is not defeasible anymore.

It can be seen that the transformation has "strengthened" the weak negation $\sim$ of normal logic programs. A defeasible rule may be applied to derive a literal only under two circumstances. One, the rule is not defeasible because the weak negation being applied on a literal p is actually the result of the literal being proven to be false through the derivation of $\neg p$. The other way in which weak negation may be used to derive a literal A is by applying CWA on the literal which is achieved by adding $\sim \neg A$ to the body of the rule.

Part IV

Inconsistent Databases

CHAPTER 9.

SP-RELATIONS

In this chapter, we present an approach to query processing for paraconsistent databases in the presence of integrity constraints. Paraconsistent databases are capable of representing positive as well as negative facts and typically operate under the open world assumption. It is easily observed that integrity constraints are usually statements about negative facts and as a result paraconsistent databases are suitable as a representation mechanism for such information. We use set-valued attributes to code large number of regular tuples into one extended tuple (with set-valued components). We define an extended relational model and algebra capable of representing and querying paraconsistent databases in the presence of integrity constraints. The extended algebra is used as the basis for query processing in such databases.

Relations that contain relations as tuple components are called non-first normal form relations [31, 59, 67]. In this paper, we restrict our attention to relations that allow only sets as tuple components and thus is a special case of non-first normal form relations [31, 59]. If the domain of an attribute is a subset of the powerset of the atomic-valued domain, we call it a set-valued attribute. The extended relational model requires that the domain of attributes be set-valued. Atomic values are represented as singleton sets. Every row in such an extended relation shall henceforth be called an **s-tuple** to differentiate it from the term **tuple** that we normally associate with regular relations and the relations will themselves be called **s-relations**. Since tuple components in s-relations are set-valued, we extend the notation to allow the "complement" operation from set theory. It will be shown that this notation, apart from increasing the clarity and simplicity of representation, also increases the power of the algebra, specifically when applied to paraconsistent relations. Thus tuple components can also contain $\overline{\{a\}}$ which represents a set containing all elements in the domain of the attribute except a, and ϕ, the empty set. Null values in s-relations are thus represented

as ϕ. $\bar{\phi}$ will now represent the entire domain. We will also make the assumption that all attributes have the same domain $\bar{\phi}$, without any loss of generality.

Consider `student(ssn,name,phone)`, a simple predicate, which describes students. In our set-valued paraconsistent relational model, we will be able to express facts such as "Student John has ssn 1234 and has two phones 1111 and 1112" using the s-tuple notation $< 1234, John, \{1111, 1112\} >$ to denote a positive fact. The functional dependency constraint $ssn \rightarrow name$ would allow us to infer negative facts in the form of the s-tuple $< 1234, \overline{\{John\}}, \overline{\phi} >$.

We now formally define set-valued extensions to the relational model. **s-tuple**.An s-tuple on Σ is any map $t : \Sigma \rightarrow \cup_{A \in \Sigma} dom(A)$, such that $t(A) \subseteq dom(A)$ for each $A \in \Sigma$. Let $\tau(\Sigma)$ denote the set of all s-tuples on Σ. Then, $\tau(\Sigma) =< \bar{\phi}, \bar{\phi}, \ldots, \bar{\phi} >$.

s-relation. An s-relation on scheme Σ is a set of s-tuples on Σ.

Figure 1 shows an s-relation in which each s-tuple has set-valued components. The ϕ in

STUDENT		
SSN	Name	Ph
{111}	{Tom}	{4046514633, 4046654321}
{888}	{Jennifer}	ϕ

Figure 9.1. An s-relation, STUDENT

the last s-tuple indicates that there is no value (NULL) under the column Ph for that s-tuple. Given an s-relation e which contains k s-tuples where the i^{th} tuple is denoted by $< t_{i1}, \ldots, t_{in} >$ where all $t_{ij}, 1 \leq i \leq k, 1 \leq j \leq n$, are set-valued, there is a one-to-one correspondence between the s-relation e and the ordinary relation corresponding to e. We denote by e_{ord} the ordinary relation corresponding to the s-relation e defined as follows:

$$e_{ord} = \cup_{i=1}^{k} t_{i1} \times \ldots \times t_{in}$$

We introduce two new operators **REDUCE** and **COMPACT**, which will be used in all operations in the model.

9.1 The Operator REDUCE

We define an operator, $REDUCE$, which takes an s-relation e on scheme Σ as input and returns another s-relation on scheme Σ after eliminating redundant s-tuples. Below is a formal description of $REDUCE$.

$REDUCE(e) = \{t_1 \in e | \neg(\exists(t_2 \in e)) \wedge \forall_{x \in \Sigma}(t_2[x] \supset t_1[x])\}$

e		
A	B	C
{1,2,3}	$\bar{\phi}$	$\overline{\{1\}}$
{3}	{6,7}	$\overline{\{1,3\}}$
{6}	{8}	{7,9}

$REDUCE(e)$		
A	B	C
{1,2,3}	$\bar{\phi}$	$\overline{\{1\}}$
{6}	{8}	{7,9}

Figure 9.2. Example of $REDUCE$

9.2 The Operator COMPACT

We introduce a new operator $COMPACT$ that takes an s-relation e as input, and produces another s-relation e'.The new s-relation e' will have atmost the number of s-tuples in e or fewer. The algorithm below will take as input an s-relation e and the associated functional dependencies and produce another s-relation e' as output.

Algorithm $COMPACT(e, R, F)$

Input: An s-relation e under the scheme R and a set of associated functional dependencies F.

Output:A compacted s-relation e'.

Method: The s-relation e' is obtained as follows:

1. Let $C = \{k_1, k_2, .., k_n\}$ be the candidate keys computed from F.
2. if$(\exists i, j | (k_i \cap k_j = \phi) \vee ((|C| = 1) \wedge (|k_1| \neq 1)))$

   return e

   else{

if($\exists k_c \in C | (|k_c| = 1) \vee (\forall_{i,j}(k_i \cap k_j = k_c) \wedge$

($|k_c| = 1$))

for every set of tuples T for which $\pi_{<R-k_c>}(T)$

is singleton,

replace them with a new s-tuple t such that

$t[R - k_c] = \pi_{<R-k_c>}(T)$ and $t[k_c] = \cup\pi_{k_c}(T)$

return the new compacted relation e'

}

The intuition behind the $COMPACT$ algorithm is that *the column picked to be set-valued would be one that belongs to all keys of the s-relation.* When the keys of the s-relation are computed, a number of scenarios can occur:

Case 1: The s-relation has only one key attribute.

Case 2: The s-relation has only one key but the key consists of more than one attribute.

Case 3: The s-relation has multiple keys and all of them have exactly one attribute in common.

Case 4: The s-relation has multiple keys and NOT all of them have exactly one attribute in common.

Let us consider the operation of $COMPACT$ on the s-relation e below. Let A be the only key attribute of the s-relation. In both cases 2 and 4 the algorithm just returns the original

e		
A	B	C
{1}	{6,7}	{9}
{2}	{3}	{8}
{3}	{6,7}	{9}

$COMPACT(e)$		
A	B	C
{1,3}	{6,7}	{9}
{2}	{3}	{8}

Figure 9.3. An example of $COMPACT$

s-relation and does not attempt to $COMPACT$ the s-relation e any further. This is because the time complexity of $COMPACT$ is exponential for more than one attribute.

Let us now examine cases 1 and 3 which ARE candidates for $COMPACT$. The simplest is Case 1 where the s-relation has just one key attribute. Since that attribute is the ONLY key of the s-relation, it very likely that we will have two or more tuples that have identical values under all other attributes (it should be noted here that this would not have been possible if there WAS another key for the s-relation). These tuples can then be combined into a single s-tuple with the key attribute set-valued.

Case 3 is where there are multiple keys in the s-relation and all have exactly one attribute in common. Since we are looking for exactly one attribute to perform $COMPACT$, an attribute that appears in all keys of the s-relation seems an ideal candidate going by the intuition that $COMPACT$ was based on.

9.3 Algebraic Operators on s-relations

Here we define the algebraic operators for s-relations. We also define an operator $REP^{\{\}}$ which takes an ordinary relation under any scheme Σ as input and produces an s-relation under the same scheme as follows:

$REP^{\{\}}(R) = \{s|(\forall t \in R)(\forall A \in \Sigma)s[A] = \{t[A]\}\}$

This operator $REP^{\{\}}$ is used in the definition of the difference operator. **Union**. The union of s-relations e_1 and e_2, under scheme Σ and with functional dependencies F_1 and F_2 repectively, denoted by $\cup^s$, is defined as follows:

$e_1 \cup^s e_2 = COMPACT(REDUCE(\{t|t \in e_1 \text{ or } t \in e_2\}), \Sigma, F_1 \cup F_2)$

Difference. The difference between two s-relations, e_1 and e_2, under scheme Σ and with functional dependencies F_1 and F_2 repectively, denoted by $-^s$, is defined as follows:

$e_1 -^s e_2 = COMPACT(REDUCE(REP^{\{\}}(e_{1ord}) \cup e_2) - e_2, \Sigma, F_1)$

Intersection. We use the identity $e_1 \cap^s e_2 = e_1 -^s (e_1 -^s e_2)$ with the algorithm above to compute $\cap^s$ on s-relations e_1 and e_2. **Selection.** The selection of e by F, where e is an s-relation on scheme Σ, denoted by $\sigma^s_F(e)$ where e is an s-relation on scheme Σ as follows:

Let $\theta = \{<, \leq, =, >, \geq, \neq\}$. Let c, c_1, c_2 be constants and $X, Y \in \Sigma$ The formula F can be

classified into one of four cases:

Case 1: $c_1 \theta c_2$

Case 2: $X \theta c$

Case 3: $c \theta Y$

Case 4: $X \theta Y$.

Case 1 is trivial and returns either TRUE or FALSE and hence the query returns either e or the empty relation.

Case 2: Without loss of generality, assume X is the first column in the s-relation e. Let $t =< u_1, u_2 \ldots u_n >$ be any s-tuple in e and let $u_1 = \{a_1, a_2 \ldots a_m\}$. Then, $u_1' = \{a_i | 1 \leq i \leq m$ and $a_i \theta c$ is true $\}$. If $u_i\prime = \phi$, then drop t. Else return $t' =< u_1', u_2 \ldots u_n >$

Case 3 is similar to Case 2.

Case 4: Without loss of generality, let X be the first column and Y be the 2^{nd} column in s-relation e. Let $t =< u_1, u_2 \ldots u_n >$ be any s-tuple in e and let $u_1 = \{a_1, a_2 \ldots a_m\}$ and $u_2 = \{b_1, b_2 \ldots b_m\}$. Let $c = b_1$ and repeat Case 2 to generate t_1'. Let $c = b_2$, $c = b_3$ and so on to generate a new s-tuple t_i' for each b_i. Thus atmost n new s-tuples are generated for each $b_i, 1 \leq i \leq n$. This can be reduced to $min(m, n)$ s-tuples by choosing $Y \theta' X$ instead of $X \theta Y$ where θ' is the complementary operation to θ.

Projection. The projection of e onto Δ, denoted by $\pi^s_\Delta(e)$ where e is an s-relation on scheme Σ, and $\Delta \subseteq \Sigma$ and F is the set of functional dependencies, is defined as follows: $\pi^s_\Delta = COMPACT(REDUCE(\{t[\Delta] | t \in e\}), \Delta, F)$. **Cartesian product**. Let e_1 and e_2 be two s-relations on schemes Σ and Δ respectively. Then, the cartesian product, denoted by $e_1 \times^s e_2$ is an s-relation on scheme $\Sigma \circ \Delta$ defined as

$e_1 \times^s e_2 = REDUCE(\{t_1 \circ t_2 | t_1 \in e_1$ and $t_2 \in e_2\})$ where $\circ$ denotes the concatenation operation.

9.4 Set-valued Paraconsistent Relations

Unlike normal relations where we only retain information that is believed to be true of a particular predicate, the paraconsistent relational model is a step towards completing the database. In a paraconsistent relation, we also retain what is *believed to be false* of a particular predicate [6, 7].

We define paraconsistent relations formally as follows:

Paraconsistent relations. A paraconsistent relation on a scheme Σ is a pair $< R^+, R^- >$ where R^+ and R^- are ordinary relations on Σ.

Thus R^+ represents the set of tuples believed to be true of R and R^- represents the set of tuples believed to be false.

We allow the paraconsistent relations to be set-valued and introduce the notion of **sp-relations**.

sp-relation. An sp-relation on a scheme Σ is a pair $< R^+, R^- >$ where R^+ and R^- are s-relations on Σ. Also,

$COMPACT(R) =< COMPACT(R^+), COMPACT(R^-) >$

$REDUCE(R) =< REDUCE(R^+), REDUCE(R^-) >$

Figure 9.4 is a instance of a set-valued paraconsistent employee database.

Employee		
SSN	Name	Age
{111}	{Navin}	{24}
{222}	{James}	{23}
{333}	{Jennifer}	{25}
{555}	$\bar{\phi}$	$\bar{\phi}$
{666}	$\bar{\phi}$	$\bar{\phi}$

Supervisor	
SSN	SuperSSN
{111}	{333}
{222}	{111}
{111}	$\overline{\{333\}}$
{333}	$\bar{\phi}$

Figure 9.4. A set-valued paraconsistent employee database

The database has a relation $Employee =< Employee^+, Employee^- >$ which represents the employee entity and the relation $Supervisor =< Supervisor^+, Supervisor^- >$ which

represents their supervisors(who are themselves employees). The tuples $< \{555\}, \bar{\phi}, \bar{\phi} >$ and $< \{666\}, \bar{\phi}, \bar{\phi} >$ in $Employee^-$ indicate that there are no employees with SSN='555' or SSN='666'.

9.5 Algebraic Operators on sp-relations

Here we define the algebraic operators for sp-relations. Let R and S be two sp-relations on scheme Σ.

Union. The union of R and S, denoted $R \cup^{sp} S$, is an sp-relation on scheme Σ, given by

$(R \cup^{sp} S)^+ = R^+ \cup^s S^+$, $(R \cup^{sp} S)^- = R^- \cap^s S^-$,

where $\cup^s$ denotes union over s-relations and $\cap^s$ denotes intersection over s-relations.

Complement. The complement of sp-relation R, denoted by $-^{sp}R$ is an sp-relation on scheme Σ,given by

$(-^{sp}R)^+ = R^-$, $(-^{sp}R)^- = R^+$

Intersection. The intersection of sp-relations R and S, denoted by $R \cap^{sp} S$, is an sp-relation on scheme Σ, given by,

$(R \cap^{sp} S)^+ = R^+ \cap^s S^+$, $(R \cap^{sp} S)^- = R^- \cup^s S^-$

Difference. The difference of sp-relations R and S, denoted by $R -^{sp} S$, is an sp-relation on scheme Σ, given by

$(R -^{sp} S)^+ = R^+ \cap^s S^-$, $(R -^{sp} S)^- = R^- \cup^s S^+$

Selection. Let R be an s-relation under scheme Σ and let F be a formula of the form $X \theta Y$ where $\theta = \{<, >, =, <=, >=, \neq\}$. Then, the selection of R by F, denoted by $\sigma_F^{sp}(R)$ is a sp-relation on Σ, given by

$\sigma_F^{sp}(R)^+ = \sigma_F^s(R^+)$, $\sigma_F^{sp}(R)^- = R^- \cup^s \sigma_{\neg F}^s(\tau(\Sigma))$

where σ^s is the selection operation on s-relations.

The negative component, $R^- \cup \sigma_{\neg F}^s(\tau(\Sigma))$, is computed as follows. Since $\tau(\Sigma)$ represents the set of all tuples on Σ, it can be represented as the single $|\Sigma|$-tuple $< \bar{\phi}, \bar{\phi}, ..., \bar{\phi} >$. Selecting s-tuples that satisfy $\neg F$ from $\tau(\Sigma)$ will thus mean removing from each component $\bar{\phi}$ in $\tau(\Sigma)$,

those values that satisfy $\neg\neg F$, or F. Notice that when F is of the form $X\theta Y$, and either X or Y is a constant, $\sigma^s_{\neg F}(\tau(\Sigma))$ will always contain only one s-tuple.

Projection. Let R be an sp-relation on scheme Σ and let $\Delta \subseteq \Sigma$. Then, the projection of R onto Δ, denoted by $\pi^{sp}_{\Delta}(R)$, is an sp-relation on Δ, given by,

$\pi^{sp}_{\Delta}(R)^+ = \pi^s_{\Delta}(R^+)$, $\pi^{sp}_{\Delta}(R)^- = \{t \in \tau(\Delta) | t^{\Sigma} \subseteq (R^-)^{\Sigma}\}$,

where π^s_{Δ} is the projection over Δ of s-relations.

The negative component of the projection denotes the set of all tuples in scheme Δ, $\tau(\Delta)$ such that all their extensions are present in $(R^-)^{\Sigma}$.

We define extensions of an s-tuple as follows:

If Σ and Δ are relation schemes such that $\Delta \subseteq \Sigma$, then for any s-tuple $t \in \tau(\Delta)$, we let t^{Σ} denote the set of $|\Sigma|$ -tuples $\{t' | t'(A) = t(A), \text{ for all } A \in \Delta \text{ and } t'(B) = \bar{\phi}, \text{ for all } B \in \Sigma - \Delta\}$.

Join. Let R and S be sp-relations on schemes Σ and Δ respectively. Then the natural join of R and S, denoted by $R \bowtie^{sp} S$, is given by,

$(R \bowtie^{sp} S)^+ = R^+ \bowtie^s S^+$, $(R \bowtie^{sp} S)^- = (R^-)^{\Sigma \cup \Delta} \cup^s (S^-)^{\Sigma \cup \Delta}$

where $\bowtie^s$ can be defined in terms of $\times^s$ and σ^s.

9.6 Representing Constraints in sp-relations

A functional dependency of the form $A \rightarrow B$ in a relation R introduces a constraint that for any two tuples $t_1, t_2 \in R$, if $t_1[A] = t_2[A]$ then $t_1[B] = t_2[B]$. This results in an explosion in the information content when the relation is paraconsistent. Whenever a functional dependency is present in a relation, the constraint thus introduced implies that we can infer a number of facts to be false in R, or, in other words, we can conclude that those facts will belong to R^-. Let $R =< R^+, R^- >$ be a paraconsistent relation under the scheme Σ. Assume that there is a tuple $t \in R^+$ with $t[A] = a$ and $t[B] = b$ and a functional dependency $A \rightarrow B$ for some attributes $A, B \in \Sigma$. This implies that any tuple with $t[A] = a$

and $t[B] \neq b$ will be in R^-. Thus R^- will contain tuples of the form

$\{t \mid (t[A] = a) \wedge (t[B] = x) \wedge (x \neq b) \wedge\ (t[\Sigma - \{A, B\}] \in \tau(\Sigma - \{A, B\}))\}$ for every FD $A \rightarrow B$.

With sp-relations, it is much easier to represent the functional dependencies in the negative component. The notations that were introduced in Section 1 now simplify the process and it involves introducing just *one s-tuple of the form* $< a, \overline{\{b\}}, \bar{\phi}, \bar{\phi}... >$ *in* R^-.
Similarly, a referential integrity constraint on a database requires that each value in the foreign key in a relation matches the value in the primary key. In paraconsistent relations, when a value is stored as false in the primary key of a relation i.e. in the negative component in all possible combinations, then all foreign keys matching that primary key value will also become false. For example, in the employee database, the employee relation with primary key SSN has values '555' and '666' stored in the negative component in all possible combinations. This implies that no employee exists with either SSN '555' or '666'. The supervisor relation has SSN as a foreign key. Since SSN values '555' and '666' are false in the employee relation, all extensions of these values can be introduced in the negative component of the supervisor relations as $< 555, \bar{\phi} >$ and $< 666, \bar{\phi} >$. The database instance of Figure 9.4 modified to include the FD $SSN \rightarrow Name, Age$ and the attribute references SSN and $SuperSSN$ in $Supervisor$ to SSN in $Employee$ is shown in Figure 9.5.

Employee		
SSN	Name	Age
{111}	{Navin}	{24}
{222}	{James}	{23}
{333}	{Jennifer}	{25}
{555}	$\bar{\phi}$	$\bar{\phi}$
{666}	$\bar{\phi}$	$\bar{\phi}$
{111}	$\overline{\{\text{Navin}\}}$	$\overline{\{24\}}$
{222}	$\overline{\{\text{James}\}}$	$\overline{\{23\}}$
{333}	$\overline{\{\text{Jennifer}\}}$	$\overline{\{25\}}$

Supervisor	
SSN	SuperSSN
{111}	{333}
{222}	{111}
{111}	$\overline{\{333\}}$
{333,555,666}	$\bar{\phi}$
$\bar{\phi}$	{555,666}

Figure 9.5. The employee database instance after coding constraints

CHAPTER 10.

SOURCE-AWARE REPAIRS FOR INCONSISTENT DATABASES

Logic programming has been used earlier in order to obtain the repairs of the database in [4, 38, 40]. The basic idea is to construct a program, called the *repair program* such that the answer sets [33] of the repair program correspond to the repairs of the database. The repair program is a disjunctive logic program with two kinds of negation, explicit negation and default negation. The problem of finding "preferred repairs", however, is relatively unexplored [16, 39, 40]. The situation where the database is both inconsistent and incomplete is common in practice. Repairing such databases has not been studied much. To our knowledge, the only known work which addresses the problem of inconsistent databases containing *null values* is [13]. It has already been shown that there can be an exponential number of repairs for an inconsistent database [26]. In this situation, it appears reasonable that we look only for a subset of the possible repairs of the database. Preference for one repair over another may be based on a number of criteria. Greco et al [39] use a function that returns a real number as the "quality" of the repair. Such a method is more appropriate in a setting where minimal number of insertions and deletions denotes a "good" repair. However, the method is general and the function can be used to express different criteria. In this chapter, we explore the specific case where along with each tuple in the database is attached information regarding which sources confirm the tuple and which sources do not. Such a data model is the IST model of Sadri [69]. The IST model was targeted towards modeling incomplete information. We propose a framework under which repairs may be computed based on a preference for a subset of the sources which we consider "reliable" thus proposing a solution to the problem of computing preferred repairs for databases that are both inconsistent and incomplete.

10.1 Motivation

Let us revisit the example database shown in Figure 5.5. Assume that we also have attached to each tuple in the database, a set of sources confirming the information provided by the tuple. Let us assume, for instance, that sources s_1 and s_2 confirm that the first tuple $(c1, p1)$ in the first table is correct and that sources s_1 and s_3 confirm that the first tuple $(c1, p2)$ in the second table is correct. Assume that similar source information is available for every tuple in the database. The repairs of the inconsistent database are shown in Figure 6.3. One contains the tuple $(c1, p1)$ and the other contains the tuple $(c1, p2)$. The other tuples appear in both repairs. If we were to assume that the information from source s_2 is more reliable than the information from source s_3, then we would prefer the first repair to the second one.

In the event of having inconsistent information, the most useful information in order to resolve the inconsistency is perhaps the source of the conflicting information. Several other criteria for resolving inconsistencies have been studied. One of the criteria in [40] is preference for certain updates over others. The criteria used in [38] is minimal updates. Resolving conflicts based on the reliability of the source of the information has not been studied much from a logic programming perspective. [46] provides a framework for incorporating source information in a deductive database and gives a semantics for such databases. Our aim in this chapter is to incorporate source information into the repair program [4] so that the answer sets of the repair program is a subset of the set of all repairs. This subset will be selected based on a preference for information coming from one source over another. The approach that we adopt in order to store source information is the IST method of Sadri [69, 70, 71].

10.2 Information Source Tracking Method

In this section, we briefly describe the information source tracking method of Sadri [69]. The reader is referred to [69] for a more detailed description. The IST method is an extension

to the relational model that permits the user to record the contributing information sources along with the data. This allows the system to calculate the probability of the validity of the tuple for each tuple that appears in the answer to a query.

The IST method uses an extended relational model. An *extended relational scheme* R is a set of attributes $\{A_1, \ldots, A_n, I\}$ where $A_1, \ldots, A_n$ are regular attributes and I is a special attribute, called the *source attribute*. Each attribute A_i has a domain of values $D_i, 1 \leq i \leq n$. The domain of the source attribute I, denoted by D_I, is the set of vectors of length k with -1,0,1 elements, that is, $D_I = \{< a_1, \ldots, a_k > |\ a_i \in \{-1, 0, 1\}, i = 1, \ldots, k\}$ where k is the number of information sources. An element of D_I is called an *source vector*.

A tuple on the extended scheme $R = \{A_1, \ldots, A_n, I\}$ is an element of $D_1 \times, \ldots, \times D_n \times D_I$. A *relation instance* r on the scheme R is a set of tuples on R. A source vector u for a tuple t identifies sources that contribute to t. Intuitively, t is valid if all sources having a $+1$ entry in u are correct, and all those having a -1 entry in u are incorrect. Usually, base relations consist only of tuples with either 0 or $+1$ in the source vector for the tuple. Source vectors containing -1 are obtained when the extended algebra operations are applied on the base relations. Here, since we are concerned with only the source vectors for tuples, the operations of the relational algebra are not defined. The reader is referred to [69] for details.

10.3 The Repair Program

We now describe how this source information can be incorporated into the repair program for the inconsistent database. Incorporating source information into a deductive database has already been studied in [46]. The intention there was to model uncertain information in a deductive database. Here we incorporate source information in a repair program in order to express preferences for some repairs over others.

The repair program is an extended disjunctive logic program. An extended logic program is one which has two forms of negation, a default negation *not* and an explicit negation $\neg$ [33]. An extended disjunctive logic program is a set of rules of the form

$$A_1 \vee \ldots \vee A_l \leftarrow B_1, \ldots, B_m, \text{ not } C_1, \ldots, \text{ not } C_n$$

where $A_1, \ldots, A_l, B_1, \ldots, B_m, C_1, \ldots, C_n$ are literals of the form P or $\neg P$.

The repair program we construct here is similar to the one described in [4]. We consider constraints of the form described in [4], the *binary integrity constraints* or BIC's. The constraints take one of the following three forms:

$$p_1(\bar{x}_1) \vee p_2(\bar{x}_2) \vee \varphi$$

$$p_1(\bar{x}_1) \vee \neg q_1(\bar{y}_2) \vee \varphi$$

$$\neg q_1(\bar{y}_1) \vee \neg q_2(\bar{y}_2) \vee \varphi$$

where $p_i(\bar{x}_i)$ and $q_j(\bar{y}_j)$ are database atoms and φ is a first order formula consisting only of built-in predicates and free variables appearing in the p_i's and q_j's .

Before constructing the repair program, we state the assumptions we make: The database has a finite domain D, the set of constraints on the database IC, are formulas of the form defined above and the data obtained from each source is consistent with IC.

The repair program for the inconsistent database consists of two sets of rules. One set of rules is called the *change program*, the part of the repair program that is responsible for the insertions and deletions in order to restore consistency. The other set of rules are the *default rules* or *persistence rules*, which enforce the fact that tuples in the database remain intact unless they violate constraints. The source information is incorporated into the logic program only in the set of default rules. For every predicate p in the database, its "primed version" p' is the new repaired version of the predicate p. The original predicate p itself remains untouched.

10.3.1 The change program

The change program is left untouched and is exactly as in [4]. We reproduce here the change program.

Definition 55. *Given a set of BIC's IC and a database instance r, the change program consists of the following rules:*

1. *(a) For every ground database atom $p(\bar{a}) \in r$, the fact $p(\bar{a})$.*

   (b) For every $a \in D$, the fact $dom(a)$.

2. *For the integrity constraints of the forms in (1), the triggering rules*

$$\begin{aligned} p_1'(\bar{X}_1) \vee p_2'(\bar{X}_2) &\leftarrow dom(\bar{X}_1, \bar{X}_2),\ not\ p_1(\bar{X}_1), \\ &\quad notp_2(\bar{X}_2), \neg\varphi. \\ p_1'(\bar{X}_1) \vee \neg q_1'(\bar{Y}_1) &\leftarrow dom(\bar{X}_1),\ not\ p_1(\bar{X}_1), \\ &\quad q_1(\bar{Y}_1), \neg\varphi. \\ \neg q_1'(\bar{Y}_1) \vee \neg q_2'(\bar{Y}_2) &\leftarrow q_1(\bar{Y}_1), q_2(\bar{Y}_2), \neg\varphi. \end{aligned}$$

3. *For an IC of the form $p_1(\bar{x}_1) \vee p_2(\bar{x}_2) \vee \varphi$, the pair of stabilizing rules*

$$\begin{aligned} p_1'(\bar{X}_1) &\leftarrow dom(\bar{X}_1), \neg p_2'(\bar{X}_2), \neg\varphi. \\ p_2'(\bar{X}_2) &\leftarrow dom(\bar{X}_2), \neg p_1'(\bar{X}_1), \neg\varphi. \end{aligned}$$

   For an IC of the form $p_1(\bar{x}_1) \vee \neg q_1(\bar{y}_2) \vee \varphi$, the pair of stabilizing rules

$$\begin{aligned} p_1'(\bar{X}_1) &\leftarrow dom(\bar{X}_1), q_1'(\bar{Y}_1), \neg\varphi. \\ \neg q_1'(\bar{Y}_1) &\leftarrow dom(\bar{Y}_1), \neg p_1'(\bar{X}_1), \neg\varphi. \end{aligned}$$

   For an IC of the form $\neg q_1(\bar{y}_1) \vee \neg q_2(\bar{y}_2) \vee \varphi$, the pair of stabilizing rules

$$\begin{aligned} \neg q_1'(\bar{Y}_1) &\leftarrow dom(\bar{Y}_1), q_2'(\bar{Y}_2), \neg\varphi. \\ \neg q_2'(\bar{Y}_2) &\leftarrow dom(\bar{Y}_2), q_1'(\bar{Y}_1), \neg\varphi. \end{aligned}$$

10.3.2 The persistence rules

The *persistence rules* are the set of rules that enforce that the ground facts remain in the database unless they violate constraints. The persistence rules are divided into three sets of rules: the *s-rules*, which incorporate source information into the repair program, the *persistence defaults*, which state that data persists unless it violates constraints, and the *starter rules* for the source-aware answer sets of the repair program.

In this section, we show how source information is used in the repair program. First we illustrate how a table in the IST method of Section 10.2 looks. Figure 10.1 is an example of

Teaches		
Class	Professor	I
c1	p1	1 1 0
c1	p2	1 0 0

Figure 10.1. An example of a table in the IST method

a table where we have the source vectors stored along with each tuple. Let us assume that the data has been collected from three sources s_1, s_2 and s_3. The source vectors indicate that sources s_1 and s_2 confirm the tuple $(c1, p1)$. In Section 10.2, we associated with each source s_i a Boolean variable f_i which indicated whether the source was correct or not. In the logic program, however, we will include a propositional constant of the same name as the source which indicates whether or not we *believe* in the source. For instance, the propositional constants s_1 and $\neg s_2$ indicate that we believe in the source s_1 and do not believe in the source s_2. We will call these literals *s-literals*. For instance, the set of s-literals for the table in Figure 10.1 are $\{s_1, s_2, s_3, \neg s_1, \neg s_2, \neg s_3\}$.

We first introduce what we call the set of *s-rules* in the logic program. We associate an s-rule with every fact in the database. For every predicate p in the database, the s-rule for each fact in the table for p will be stored with a suffix s, that is, p_s. However, the s-rules differ from the normal ground facts in the database in that they are *conditional facts*, i.e.,

they are true if certain conditions hold. A similar idea has been explored in [75]. There, conditional facts were introduced in order to simulate disjunctions through a normal logic program. Here, we introduce conditional facts to reflect our beliefs in data sources. For instance, the s-rule for the first tuple in the $Teaches$ table shown in Figure 10.1 is as follows:

$$Teaches_s(c1, p1) \leftarrow s_1, s_2.$$

The bodies of s-rules consist only of s-literals. For instance, the first s-rule here says that we must include $Teaches_s(c1, p1)$ in every repair of the database if we *believe* in the sources s_1 and s_2. Thus the heads of the s-rules are conditional in the sense that may or may not appear in every repair of the database depending on a choice of sources to be believed. The s-rules are stored in the logic program in addition to the regular rules to store the facts.

The *persistence defaults* enforce the fact that data persists unless it violates some constraints. The persistence defaults for every predicate p in the database are now written as follows:

$$\begin{aligned} p'(\bar{X}) &\leftarrow p_s(\bar{X}). \\ p'(\bar{X}) &\leftarrow p(\bar{X}),\ \text{not}\ \neg p'(\bar{X}). \\ \neg p'(\bar{X}) &\leftarrow dom(\bar{X}),\ \text{not}\ p(\bar{X}),\ \text{not}\ p'(\bar{X}). \end{aligned}$$

The first rule here is in effect the only addition to the repair program. This rule states that every fact that becomes true through the s-rules must be included in every repair of the database. The remaining rules are the same as those in the repair program of [4]. This is done so that the source information must be used *only in order to resolve inconsistencies* and repairs that do not contain facts generated using source information must still be produced even in the presence of source information.

For every source s_i, we introduce the pair of *starter rules*

$$s_i \leftarrow s_i. \quad \neg s_i \leftarrow \neg s_i.$$

These rules are included simply to reproduce the s-literals chosen by the user in each answer set.

10.4 Source-aware Answer Sets of the Repair Program

We now explain how the answer sets are constructed for the repair program. The stable model semantics was introduced by Gelfond and Lifschitz in [32] and later extended to extended logic programs and disjunctive logic programs in [33]. The stable model semantics is arguably the most widely accepted semantics for logic programs. The stable models of extended logic programs are called *answer sets*. In [4], a slight variation of the stable model semantics was used in order to compute the repairs. There, the repair program was treated as a disjunctive logic program with exceptions [44]. The idea is straightforward: We believe in a ground fact $p(a_1, \ldots, a_n)$ unless an exception to it has been generated by the presence of $\neg p(a_1, \ldots, a_n)$. The stable models of logic programs with exceptions are called *e-answer sets*. It has been shown in [44] that there is a one-to-one correspondence between the answer sets and the e-answer sets of a program. Hence, we can talk of them interchangeably. The reader is referred to [44] for a detailed description of e-answer sets and to [4] for the e-answer sets of the repair program. It has been shown in [4] that there is a one-to-one correspondence between the e-answer sets of the repair program and the minimal repairs of the database.

The answer set semantics is based a certain transformation defined on interpretations. We will not describe the semantics here. The reader is referred to [33] for details. We explain the construction of the repairs of the program described in the last section. The idea here is to obtain the repairs of the database based on a choice of s-literals by the user. From the set of all possible repairs, we want to be able to ignore those repairs that contain tuples that

are not confirmed by the sources that we believe in when such tuples are in conflict with source confirmed tuples. In order to achieve this, we will propose a minor reformulation of the answer set semantics for the s-rules alone.

The answer set reformulation is as follows: Let Π be the repair program and let $HB(\Pi)$ denote the Herbrand base of Π. Let S_l be the set of s-literals of Π. Let $S \subset HB(\Pi)$ and s_{lits} is a non-empty subset of S_l. Let $S_{source} = S \cup s_{lits}$. The set s_{lits} denotes the set of sources we want to believe(disbelieve).

Definition 56. *The transformation,* $\Pi_{S_{source}}$ *of* Π *w.r.t* S_{source} *is obtained by:*

1. *Deleting every rule with* $not\ L$ *in the body with* $L \in S_{source}$ *and deleting every s-rule that:*
   (a) *has* $\neg s$ *in the body with* $s \in s_{lits}$ *OR*
   (b) *does not have every literal from* s_{lits} *in its body*
2. *Deleting the negative literals from the bodies of the remaining rules and deleting every literal from the bodies of the remaining s-rules*

We now obtain a positive logic program $\Pi_{S_{source}}$. S_{source} *is a source-aware answer set of* Π *if it is the least model of* $\Pi_{S_{source}}$.

Observe that the set s_{lits} denoting our choice of sources to believe or disbelieve implicitly denotes a conjunction of those literals. This transformation relies on the assumption that sources by themselves are consistent. Hence it follows that a conjunction of the beliefs of a set of sources is also consistent.

$$\text{Let } Bel(s_i) = \{t \mid t \text{ is confirmed by the source } s_i\}$$

Let $s_{lits} = \{s_1, \ldots, s_n\}$. This denotes the set of tuples

$$\bigcap_{i=1}^{n} Bel(s_i)$$

which is the set of tuples confirmed by *every* one of $s_1, \ldots, s_n$.

Notice that the user might also believe a tuple if it is confirmed by *any* of $s_1, \ldots, s_n$. This is the set

$$\bigcup_{i=1}^{n} Bel(s_i)$$

Notice that the latter formula may represent an inconsistent set of tuples. The repair program discussed here does not support such a choice of s-literals by the user. However, this can be accomplished by constructing n sets of repairs. For instance, if the user chose to believe tuples confirmed either by the source s_1 or the source s_3, then we run the repair program once with $s_{lits} = \{s_1\}$. The answer sets of this program will correspond to all the repairs based on a belief in s_1. Next, we run the repair program again but this time we set $s_{lits} = \{s_3\}$. Now the set of repairs obtained is based a belief in the source s_3. The union of these two sets of repairs will correspond to the belief in source s_1 or s_3.

Theorem 10.4.1. *1. For every source-aware answer set S_{source} of Π, there exists a repair r' of the database instance r w.r.t the integrity constraints IC such that $r' = \{p(\bar{a}) \mid p'(\bar{a}) \in S_{source}\}$*

2. For every repair r' of the database instance r w.r.t the integrity constraints IC that is consistent with the set of sources believed(disbelieved), there exists a source-aware answer set S_{source} such that $r' = \{p(\bar{a}) \mid p'(\bar{a}) \in S_{source}\}$

Proof. The proof of the theorem is straightforward and follows from Theorem 1 of [4]. Let us denote the repair program of [4] as Π_r and its answer sets (the repairs of the database) as S. Let us denote the set of s-rules in our repair program Π as P_s. Then, it can be seen that $\Pi = \Pi_r \cup P_s$ (the other rules of the form $s_i \leftarrow s_i$ are used to simply reproduce them in the answer sets and have no effect on the repairs). Let us denote by S' the answer sets of Π (which are simply the source-aware repairs). We have to show that the addition of the s-rules to the original repair program simply has the effect of removing those repairs that do not include the user-chosen s-rules (based on the chosen s-literals). If the subset of P_s

chosen by the user appears in every repair of Π_r then the addition of P_s to Π_r has no effect on the answer sets of the program by our reformulation of the answer set semantics and we have $S = S'$. Now let us assume that there exist some answer sets in S that do not contain the user-chosen facts. This implies that these facts conflict with some other fact(s) from the original persistence rules, i.e., the facts of the form $p(\bar{X})$. (the conflict cannot be within P_s since we assume that each source by itself is consistent). Since the user-chosen facts are forced into every answer set by the new persistence rule $p'(\bar{X}) \leftarrow p_s(\bar{X})$, the answer sets of Π_r that do not contain the user-chosen facts are no longer answer sets of Π. The other answer sets of Π_r are still answer sets of Π since the rest of the program is unchanged. Hence we have that $S' \subseteq S$. □

10.5 An Illustration

In this section, we illustrate the construction of the repairs of an inconsistent database where the information is collected from 3 sources that are each consistent with the integrity constraints by themselves. However, we obtain inconsistent information when the data collected from the sources is combined. We consider here a relation $p(X, Y)$ collected from 3 independent sources s_1, s_2 and s_3. The data is subjected to a functional dependency integrity constraint $IC : X \rightarrow Y$. The data coming from each source is shown in Figure 10.2. This

s_1

P	
X	Y
a	b
c	d

s_2

P	
X	Y
a	e
b	c

s_3

P	
X	Y
b	d
c	d

Figure 10.2. Data collected from independent sources s_1, s_2 and s_3

information from the three sources may be integrated in the IST model approach to obtain the extended relation shown in Figure 10.3. The new attribute I in the table in Figure 10.3 is used to store the source vector information for each tuple. Here, the tuple (a, b) is confirmed

P		
X	Y	I
a	b	1 0 0
c	d	1 0 0
a	e	0 1 0
b	c	0 1 0
b	d	0 0 1
c	d	0 0 1

Figure 10.3. The integrated database along with source information

by source s_1 and hence the source vector associated with this tuple is $< 1, 0, 0 >$. Similarly, the tuple (c, d) is confirmed by both sources s_1 and s_3 and hence it appears twice in the extended relation, once with the source vector $< 1, 0, 0 >$ and once with the source vector $< 0, 0, 1 >$. Notice that the information is now inconsistent with the functional dependency constraint.

The repair program for this database is as follows:

The change program:

Facts:

$$p(a, b) \leftarrow . \quad p(a, e) \leftarrow . \quad p(b, c) \leftarrow .$$
$$p(b, d) \leftarrow . \quad p(c, d) \leftarrow .$$

Triggering rule:

$$\neg p'(X, Y) \vee \neg p'(X, Z) \leftarrow p(X, Y), p(X, Z), Y \neq Z.$$

Stabilizing rule:

$$\neg p'(X, Z) \leftarrow p(X, Y), ydom(Z), Y \neq Z.$$

We will assume that predicates $xdom(X)$ and $ydom(Y)$ are available that define the active domain of the database.

Persistence rules:

s-rules:

$$p_s(a, b) \leftarrow s_1. \quad p_s(a, e) \leftarrow s_2. \quad p_s(b, c) \leftarrow s_2.$$
$$p_s(b, d) \leftarrow s_3. \quad p_s(c, d) \leftarrow s_1, s_3.$$

Persistence defaults:

$$\begin{aligned} p'(X, Y) &\leftarrow p_s(X, Y). \\ p'(X, Y) &\leftarrow p(X, Y), \text{ not } \neg p'(X, Y). \\ \neg p'(X, Y) &\leftarrow xdom(X), ydom(Y), \text{ not } p(X, Y), \\ &\qquad \text{not } p'(X, Y). \end{aligned}$$

Starter rules: For $1 \leq i \leq 3$,

$$s_i \leftarrow s_i. \quad \neg s_i \leftarrow \neg s_i.$$

The s-rules for the table in Figure 10.3 brings out a subtle point. Notice that the tuple (c, d) is confirmed independently by the sources s_1 and s_3. However, in the s-rules, this is encoded as the single rule $p_s(c, d) \leftarrow s_1, s_3$. This could have also been encoded as two separate rules

$p_s(c,d) \leftarrow s_1$ and $p_s(c,d) \leftarrow s_3$. However, such an encoding will lead to incorrect results by our reformulation of the answer sets. Consider, for example the second encoding (with two rules) and let $\{s_1, s_3\}$ be the beliefs of the user. Then, by our transformation both the rules for the tuple (c,d) will be deleted. However, since (c,d) is confirmed by both s_1 and s_3 it should appear in every repair for those beliefs. This is why it is combined into the single rule $p_s(c,d) \leftarrow s_1, s_3$. Now, we obtain correct answers when the sources believed are both s_1 and s_3. Notice that if either one of the sources is chosen, then also the transformation ensures that the tuple (c,d) appears in every repair.

Let us assume that the user wishes to believe in the source s_1 since from experience he finds this source reliable.

The set of repairs based on a belief in source s_1 are shown in Figure 10.4. The set of all

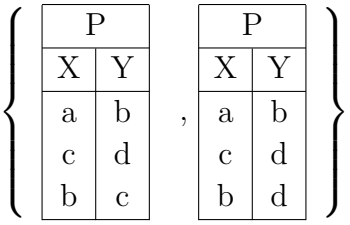

Figure 10.4. The repairs of the database based on a belief in source s_1

minimal repairs of the database are shown in Figure 10.5. Since the user chooses to believe

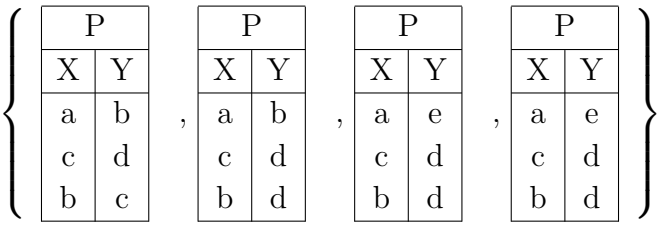

Figure 10.5. The set of all minimal repairs of the database

in the source s_1, the conflict between the tuples (a,b) and (a,e) is resolved by choosing (a,b) since this tuple is confirmed by the source s_1 whereas (a,e) is confirmed by s_2. The tuple

(c, d) is not involved in any conflict and hence appears in every repair. The conflict between tuples (b, c) and (b, d) leads to the two repairs, one containing (b, c) and the other containing (b, d).

10.6 Further Extensions

In this section, we suggest two extensions of the model that we have proposed for incorporating source information while computing repairs.

The first extension is a generalization of the proposed method. Here, instead of having $s_{lits} \subset S_l$, we have a partial ordering of the s-literals based on the priorities that we have for the sources. For instance, we may have that $s_1 < \neg s_3, s_2 < s_4$ which indicates that we prefer believing s_1 over disbelieving s_3 and we prefer believing s_2 over s_4 ($s_i < s_j$ means that s_i is preferred to s_j). Such an approach is a natural generalization of the method proposed in this chapter. Such a partial ordering translates into a prioritized logic program where the rules are prioritized based on the ordering. In our case, the ordering on the sources will lead to a prioritization of the s-rules. The priorities among the s-rules will be used in order to resolve conflicts in the database. Hence we will obtain a set of repairs in which some repairs have higher priority than others because the facts in it were derived from rules with higher priority. Prioritized logic programs have been widely studied. A notion of preferred answer sets for prioritized logic programs is discussed in [15]. Prioritized updates have been studied in [40]. Here, we assume that there are restrictions on what kind of updates may be performed on the database. This is incorporated into the repair program through *repair constraints*. Then, it is generalized to the case where there are priorities on the updates. This translates naturally into priorities on repairs.

The second extension that we propose is based on the IST method. In [69], Sadri proposes a reliability calculation based on the probability of the source being correct. Based on the source vector information for each tuple and the probability of the source being correct, we can calculate a reliability value for the tuple t, denoted by $re(t)$. Let us assume that the

reliability for every tuple in the table can be calculated. This allows us to define a notion of *repairs that can be relied upon to a certain degree.* This degree is based on the reliabilities of the tuples in the repair. Such a notion is very useful in order to attach a confidence value to a repair and hence a confidence value to answers obtained for a particular query. Let $0 < k \leq 1$.

Definition 57. *A repair r' of a database instance r is called a k-consistent repair of the database iff* $(\forall t)(t \in r' \rightarrow re(t) \geq k)$

Definition 58. *A tuple t is called a k-consistent answer to a query Q if it appears in every k-consistent repair r' of the database instance r, i.e., the answers to query Q is given by*

$$Ans(Q) = \{t \mid \quad (\forall r')(r' \text{ is a k-consistent repair of } r \rightarrow t \in r')\}$$

If we assume all sources to be 100% reliable, we then obtain every repair as a k-consistent repair. This turns out to be the special case where Definition 58 returns all consistent answers to query Q.

CHAPTER 11.
CONCLUSIONS AND FUTURE WORK

In this dissertation firstly we have presented data models that can handle incomplete information under both the OWA and CWA. Both these data models have the property of being strong representation systems in the sense that any set of instances can be represented in the systems. The second part of this dissertation focused on negation and nonmonotonic reasoning in databases and logic programs. In the first half of this part we have shown how a nonmonotonic reasoning component can be added to a data model that operates under the OWA. This makes a case for nonmonotonic reasoning under the OWA by demonstrating that useful information may be derived through this component. The other half of this part is a treatment of default negation in extended logic programs. Current semantics for extended logic programs treat default negation just as it is treated in general logic programs. As a result a number of extended logic programs are declared contradictory. We present an alternative definition of default negation in extended logic programs which behaves differently only when contradictory information may be derived. This is achieved through a translation of the extended logic program to a normal logic program. The third part of this dissertation is focused on inconsistent information in databases. First we present a data model under the OWA that handles inconsistencies by representing the negative information derived from the constraint in the database. We present an algebra for query processing with this model. Finally, we investigate the problem of computing the repairs of an inconsistent database. Extended logic programs with disjunctions have been used for computing repairs under the stable model semantics. We present a method by which lineage information can be incorporated into the repair logic program so that the number of repairs computed is reduced to a large extent.

The repair problem for inconsistent databases is of prime importance. An interesting area of future work would be to study the concept of repairs in a database that also contains

incomplete information. Both inconsistency and incompleteness are commonly seen problems when large amounts of data are integrated. This work involves two parts. First we must provide a definition of what constitutes a repair of a set of database instances some of which violate constraints. Secondly, efficient techniques for consistent query answering should be developed either by a procedure that computes all the repairs or by query rewriting.

Another area of future work would be to investigate the use of OWA models for handling inconsistencies in databases. Such models appear to be particularly useful in situations where incompleteness is present in the form of disjunctive information.

Thirdly, it may be interesting to study how repairs may be represented using some of the strong representation systems developed to handle incomplete information. Repairs are closely related to incomplete databases since essentially both can be seen as sets of instances. Leveraging some of the techniques developed for handling incomplete information may improve the efficiency of consistent query answering.

BIBLIOGRAPHY

[1] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[2] José Júlio Alferes, Phan Minh Dung, and Luís Moniz Pereira. Scenario semantics of extended logic programs. In *LPNMR*, pages 334–348, 1993.

[3] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *PODS '99*, pages 68–79, New York, NY, USA, 1999. ACM.

[4] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *Theory Pract. Log. Program.*, 3(4):393–424, 2003.

[5] Ofer Arieli. Paraconsistent declarative semantics for extended logic programs. *Annals of Mathematics and Artificial Intelligence*, 36:381–417, 2002.

[6] Rajiv Bagai and Rajshekhar Sunderraman. A paraconsistent relational data model. *International Journal of Computer Mathematics*, 55(3), 1995.

[7] Rajiv Bagai and Rajshekhar Sunderraman. Bottom-up computation of the fitting model for general deductive databases. *Journal of Intelligent Information Systems*, 6(1):59–75, January 1996.

[8] Chitta Baral and V. S. Subrahmanian. Dualities between Alternative Semantics for Logic Programming and Non-monotonic Reasoning. In Anil Nerode, Wiktor Marek, and V. S. Subrahmanian, editors, *Logic Programming and Non-Monotonic Reasoning, Proceedings of the first International Workshop*, pages 69–86. MIT Press, 1991.

[9] Chitta Baral and V. S. Subrahmanian. Stable and extension class theory for logic programs and default logics. *J. Autom. Reasoning*, 8(3):345–366, 1992.

[10] N. D. Belnap. A useful four-valued logic. In G. Eppstein and J. M. Dunn, editors, *Modern Uses of Many-valued Logic*, pages 8–37. Reidel, Dordrecht, 1977.

[11] H. A. Blair and V. S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science*, 68:135–154, 1989.

[12] Patrick Bosc and Olivier Pivert. About projection-selection-join queries addressed to possibilistic relational databases. *IEEE T. Fuzzy Systems*, 13(1):124–139, 2005.

[13] Loreto Bravo and Leopoldo Bertossi. Semantically correct query answers in the presence of null values. In *In Pre-Proc. EDBT WS on Inconsistency and Incompleteness in Databases (IIDB 06*, pages 33–47. Springer, 2006.

[14] Gerhard Brewka. Well-founded semantics for extended logic programs with dynamic preferences. *Journal of Artificial Intelligence Research*, 4:19–36, 1996.

[15] Gerhard Brewka and Thomas Eiter. Preferred answer sets for extended logic programs. *Artif. Intell.*, 109(1-2):297–356, 1999.

[16] Luciano Caroprese, Sergio Greco, Irina Trubitsyna, and Ester Zumpano. Preferred generalized answers for inconsistent databases. In *ISMIS*, pages 344–349, 2006.

[17] Edward P. F. Chan. A possible world semantics for disjunctive databases. *Knowledge and Data Engineering*, 5(2):282–292, 1993.

[18] Jui-Shang Chiu and Arbee L. P. Chen. An exploration of relationships among exclusive disjunctive data. *IEEE Trans. Knowl. Data Eng.*, 7(6):928–940, 1995.

[19] Jan Chomicki and Jerzy Marcinkowski. Minimal-change integrity maintenance using tuple deletions. *Inf. Comput.*, 197(1/2):90–121, 2005.

[20] K. L. Clark. Negation as failure. In M. L. Ginsberg, editor, *Readings in Nonmonotonic Reasoning*, pages 311–325. Kaufmann, Los Altos, CA, 1987.

[21] E. F. Codd. Missing information (applicable and inapplicable) in relational databases. *SIGMOD Rec.*, 15(4):53–53, 1986.

[22] E.F. Codd. A relational model for large shared data banks. *Comm. of the ACM*, 13(6):377–387, 1970.

[23] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *Proc. 30th VLDB Conf.*, pages 864–875, 2004.

[24] Anish Das, Omar Benjelloun, Alon Halevy, and Jennifer Widom. Working models for uncertain data. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 7, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Phan Minh Dung. An argumentation-theoretic foundations for logic programming. *J. Log. Program.*, 22(2):151–171, 1995.

[26] Thomas Eiter, Michael Fink, Gianluigi Greco, and Domenico Lembo. Repair localization for query answering from inconsistent databases. *ACM Trans. Database Syst.*, 33(2):1–51, 2008.

[27] Ronald Fagin and Moshe Y. Vardi. The theory of data dependencies - an overview. In *Proceedings of the 11th Colloquium on Automata, Languages and Programming*, pages 1–22, London, UK, 1984. Springer-Verlag.

[28] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. Conditional functional dependencies for capturing data inconsistencies. *ACM Trans. Database Syst.*, 33(2):1–48, 2008.

[29] M. Fitting. A Kripke-Kleene semantics for logic programs. *Journal of Logic Programming*, 4:295–312, 1985.

[30] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 15(1):32–66, 1997.

[31] Jaeschke G. and Schek H. Remark on the algebra of non first normal form relation. In *Proceedings of ACM Symposium on Principles of Database Systems*, 1982.

[32] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th ICSLP*, pages 1070–1080, Seattle, WA, August 1988.

[33] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3/4):365–386, 1991.

[34] John Grant. Null values in a relational data base. *Inf. Process. Lett.*, 6(5):156–157, 1977.

[35] John Grant and Jack Minker. Answering queries in indefinite databases and the null value problem. *Advances in Computing Research*, 3:247–267, 1986.

[36] John Grant and V. S. Subrahmanian. Reasoning in inconsistent knowledge bases. *IEEE Trans. Knowl. Data Eng.*, 7(1):177–189, 1995.

[37] John Grant and V.S. Subrahmanian. Applications of paraconsistency in data and knowledge bases. *Synthese*, 125:121–132, 2000.

[38] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In *Proceedings of the 17th ICLP*, pages 348–364, London, UK, 2001. Springer-Verlag.

[39] Gianluigi Greco, Sergio Greco, and Ester Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE TKDE*, 15(6):1389–1408, 2003.

[40] Sergio Greco and Ester Zumpano. Computing repairs for inconsistent databases. In *CODAS '01*, page 30, Washington, DC, USA, 2001. IEEE Computer Society.

[41] Tomasz Imieliński and Jr. Witold Lipski. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[42] ao Alcântara Jo Carlos Viegas Damásio, and Luís Moniz Pereira. Paraconsistent logic programs. In *JELIA '02: Proceedings of the European Conference on Logics in Artificial Intelligence*, pages 345–356, London, UK, 2002. Springer-Verlag.

[43] Phokion G. Kolaitis. The expressive power of stratified logic programs. *Inf. Comput.*, 90(1):50–66, 1991.

[44] Robert A. Kowalski and Fariba Sadri. Logic programs with exceptions. In *Logic programming*, pages 598–613, Cambridge, MA, USA, 1990. MIT Press.

[45] Laks V. S. Lakshmanan, Nicola Leone, Robert Ross, and V. S. Subrahmanian. ProbView: a flexible probabilistic database system. *ACM Transactions on Database Systems*, 22(3):419–469, 1997.

[46] Laks V. S. Lakshmanan and Fereidoon Sadri. Modeling uncertainty in deductive databases. In *DEXA '94*, pages 724–733, London, UK, 1994. Springer-Verlag.

[47] Leonid Libkin and Limsoon Wong. Semantic representations and query languages for or-sets. In *PODS*, pages 37–48, 1993.

[48] Ken-Chih Liu and Rajshekhar Sunderraman. Indefinite and maybe information in relational databases. *ACM Trans. Database Syst.*, 15(1):1–39, 1990.

[49] Ken-Chih Liu and Rajshekhar Sunderraman. A generalized relational model for indefinite and maybe information. *IEEE Trans. Knowl. Data Eng.*, 3(1):65–77, 1991.

[50] J. W. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[51] E. L. Lozinskii. Plausible world assumption. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *KR'89: Proc. of the First International Conference on Principles of*

Knowledge Representation and Reasoning, pages 266–275. Kaufmann, San Mateo, CA, 1989.

[52] David Maier. *Theory of Relational Databases*. Computer Science Pr, 1983.

[53] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.

[54] Jack Minker. On indefinite databases and the closed world assumption. In *Proceedings of the 6th Conference on Automated Deduction*, pages 292–308, London, UK, 1982. Springer-Verlag.

[55] Jack Minker and Carolina Ruiz. Semantics for disjunctive logic programs with explicit and default negation. *Fundamenta Informaticae*, 20(1/2/3):145–192, 1994.

[56] Robert C. Moore. Semantical considerations on nonmonotonic logic. *Artif. Intell.*, 25(1):75–94, 1985.

[57] Shamim A Naqvi. Negation as failure for first-order queries. In *PODS '86: Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 114–122, New York, NY, USA, 1986. ACM Press.

[58] Adegbemiga Ola. Relational databases with exclusive disjunctions. In *ICDE*, pages 328–336, 1992.

[59] G. Ozsoyoglu, Z. M. Ozsoyoglu, and V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. *ACM Trans. on Database Syst.*, 12(4):566–592, 1987.

[60] Luís Moniz Pereira and José Júlio Alferes. Well founded semantics for logic programs with explicit negation. In *ECAI '92: Proceedings of the 10th European conference on Artificial intelligence*, pages 102–106, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[61] Henri Prade. Lipski's approach to incomplete information data bases restated and generalized in the setting of zadeh's possibility theory. *Inf. Syst.*, 9(1):27–42, 1984.

[62] Teodor C. Przymusinski. Extended stable semantics for normal and disjunctive programs. pages 459–477, 1990.

[63] Teodor C. Przymusinski. On the declarative and procedural semantics of logic programs. *Journal of Automated Reasoning*, 5:167–205, 1995.

[64] R. Reiter. On closed world data bases. In *Readings in nonmonotonic reasoning*, pages 300–310. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[65] Kenneth A. Ross. Modular acyclicity and tail recursion in logic programs. In *PODS '91: Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 92–101, New York, NY, USA, 1991. ACM.

[66] Kenneth A. Ross and Rodney W. Topor. Inferring negative information from disjunctive databases. *J. Autom. Reason.*, 4(4):397–424, 1988.

[67] Mark A. Roth, Henry F. Korth, and Abraham Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Trans. Database Syst.*, 13(4):389–417, 1988.

[68] Domenico Saccà and Carlo Zaniolo. Stable models and non-determinism in logic programs with negation. In *PODS*, pages 205–217, 1990.

[69] Fereidoon Sadri. Reliability of answers to queries in relational databases. *IEEE TKDE*, 3(2):245–251, 1991.

[70] Fereidoon Sadri. Information source tracking method: Efficiency issues. *IEEE TKDE*, 7(6):947–954, 1995.

[71] Fereidoon Sadri. Integrity constraints in the information source tracking method. *IEEE Trans. Knowl. Data Eng.*, 7(1):106–119, 1995.

[72] Chiaki Sakama. Possible model semantics for disjunctive databases II (extended abstract). In *Logic Programming and Non-monotonic Reasoning*, pages 107–114, 1990.

[73] Chiaki Sakama. Extended well-founded semantics for paraconsistent logic programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 592–599, ICOT, Japan, 1992. Association for Computing Machinery.

[74] Marek A. Suchenek and Rajshekhar Sunderraman. On reasoning from closed world databases with disjunctive views. In *LPNMR*, pages 132–149, 1990.

[75] Rajshekhar Sunderraman. Deductive databases with conditional facts. In *BNCOD*, pages 162–175, 1993.

[76] Rajshekhar Sunderraman. Modeling negative and disjunctive information in relational databases. In *DEXA '97: Proceedings of the 8th International Conference on Database and Expert Systems Applications*, pages 337–346, London, UK, 1997. Springer-Verlag.

[77] J. Ullman. Assigning an appropriate meaning to database logic with negation. Technical Report 1994-15, Stanford Infolab, 1994. A corrected version of a paper that appeared in "Computers as Our Better Partners" (H. Yamada, Y. Kambayashi, and S. Ohta, eds.) pp. 216–225, World Scientific, Singapore, 1994.

[78] Kumar V. Vadaparty and Shamim A. Naqvi. Using constraints for efficient query processing in nondeterministic databases. *IEEE Trans. Knowl. Data Eng.*, 7(6):850–864, 1995.

[79] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.

[80] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *PODS '89: Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 1–10, New York, NY, USA, 1989. ACM.

[81] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):621–650, 1991.

[82] Lu Zhang and Ken-Chih Liu. Towards a relational model for exclusively disjunctive information. In *CSC '93: Proceedings of the 1993 ACM conference on Computer science*, pages 143–150, New York, NY, USA, 1993. ACM Press.

APPENDIX:

RELATED POSTERS/PAPERS

1. Navin Viswanath and Rajshekhar Sunderraman. Query processing in paraconsistent databases in the presence of integrity constraints. In Proceedings of SEKE, pages 580 - 585, 2007.

2. Navin Viswanath. Explicit and default negation relational databases and logic programs. In Proceedings of the SIGMOD Workshop on Innovative Database Research, IDAR, 2008.

3. Navin Viswanath and Rajshekhar Sunderraman. Defaults in open world relational databases. In Proceedings of IPMU, pages 212 - 219, 2008.

4. Navin Viswanath and Rajshekhar Sunderraman. Degrees of exclusivity in disjunctive databases. In Proceedings of ISMIS, pages 375 - 380, 2008.

5. Navin Viswanath and Rajshekhar Sunderraman. Handling disjunctions in open world relational databases. In Proceedings of NAFIPS, pages 1 - 8, 2008.

6. Navin Viswanath and Rajshekhar Sunderraman. A paraconsistent relational data model. In Handbook of Research on Innovations in Database Technologies and Applications : Current and Future Trends, pages 18 - 27. Idea Group Publishing, 2009.

7. Navin Viswanath and Rajshekhar Sunderraman. Source-aware repairs for inconsistent databases. In Proceedings of DBKDA, pages 125 - 130, 2009.

8. Navin Viswanath and Rajshekhar Sunderraman. Handling inconsistencies in extended logic programs through program transformation. Accepted for publication at IICAI 2009.