Disjunctive Deductive Databases: Semantics, Updates, and Architecture *

J.A. Fernández[3], J. Gryz[1], and J. Minker[1,2]

[1] Department of Computer Science
[2] Institute for Advanced Computer Studies
University of Maryland, College Park, Maryland 20742
[3] Bull Information Systems
Phoenix, Arizona

Abstract. The basic assumption in relational and deductive databases is that there are no gaps in our knowledge. That is, the database cannot store data that contain null values or data that is indefinite. In practical situations knowledge is not precise, and there are gaps in our knowledge. These gaps may be due to null values in the data, may arise when we combine several databases that lead to inconsistent theories, or may occur because information is indefinite in nature, such as in military or medical applications.
In this paper we describe semantics for disjunctive deductive databases that extend the work in deductive databases, solve the view update problem, and permit indefinite data to be represented efficiently. Hierarchic, stratified, and normal stable models of disjunctive databases are described. An architecture is proposed for a disjunctive deductive database system and a class of theories for which the architecture will be effective is discussed.

1 Introduction

In many knowledge base applications, information is indefinite. It is important that a user be able to query such systems and obtain information that may either be definite or indefinite. Some knowledge bases respond to queries with indefinite information with the answer *unknown* because the representation of data in such systems is based on *definite* knowledge base systems. A theory is needed that permits such knowledge to be represented, stored, and manipulated. Such a theory would permit more informative responses than *unknown*. *Disjunctive deductive databases (DDDBs)* address this problem. Specifically, we shall summarize results that provide a semantics for *DDDBs*, and that permit such databases to be updated.

Deductive databases (DDBs) are based on *logic programming (LP)* and hence, *database (DB)* theory borrows heavily from the field of *LP*. Logic programs try to find a single answer to a query and permit function symbols as arguments of predicates. *DDBs* do not permit function symbols and attempt to find all answers to a query. In *DDBs* and *DDDBs*, we deal with finite structures and use bottom up strategies to find all answers to queries.

* Invited paper for BISAFI95.

The focus of this paper is DDDBs. By a *DDDB* we mean a database where clauses are of the form

$$A_1 \vee \ldots \vee A_k \leftarrow B_1, \ldots, B_n, not \;\; D_1, \ldots, not \;\; D_m$$

where the $A_i, 1 \leq i \leq k$, the $B_j, 1 \leq j \leq n$, and $D_l, 1 \leq l \leq m$ are atoms and $m, n, k \geq 0$. By *not* , we mean a rule of default for negation [Rei78, LMR92]. When $k \neq 0$ and ($n \neq 0$ or $m \neq 0$) the clause is said to be in the *Intensional Data Base (IDB)*. If $k \neq 0$ and $n = 0$ and $m = 0$ then the clause is said to be in the *Extensional Data Base (EDB)*. All clauses are assumed to be range restricted, i.e., every variable that appears in the head of a clause, namely in the atoms of

$$A_1 \vee \ldots \vee A_k,$$

must also occur in the positive part of the body of the clause, namely,

$$B_1, \ldots, \ldots, B_n.$$

Thus, the *EDB* is a set of positive ground clauses.

A query is a clause of the form

$$\leftarrow Q_1, \ldots, Q_m$$

where $m > 0$, and the Q_i may be atoms or atoms preceded by the operator *not* .

Whereas in *Datalog* an answer to an atomic query, say $Q(X)$, is a set of constants that satisfy the query, in a *DDDB* an answer may be either a constant or a disjunction of constants that satisfy the query. Hence, informally, an answer to a query, $Q(X)$, may be $\{c \vee d\}$, where it is meant that $Q(c) \vee Q(d)$ is satisfied by the database. A formal definition of an answer is given in Section 4.

The organization of the paper is as follows. Section 2 provides some results in *LP* and in *DDDBs*. Section 3 provides semantics for alternative classes of *DDDBs*. Section 4 describes how to compute over *DDDBs* using the concept of a *model tree* ([FM91]). Section 5 reviews semantics developed for view updates in *DDDBs*. Section 6 describes a proposed architecture for a *DDDB* and discusses a class of problems for which the architecture will be effective. Section 7 provides conclusions.

2 Background

A historical perspective of the field of *DDBs* up to approximately 1988 is given in [Min88]. We are concerned here with *DDDBs* and refer the reader to that paper for details and the influence of *LP* on those developments. See also [Ull88a, Ull88b] for additional results in *DDBs*.

Work in disjunctive theories was pursued seriously starting in 1986 [Min86, Min89]. However, *DDDBs* started approximately in 1982 with the appearance of the paper [Min82], which described how to compute answers both to positive and negated queries. A historical perspective of disjunctive logic programming and *DDDBs* is given in [Min88]. While *DDBs* have a single minimal Herbrand model that characterizes the meaning of the database, *DDDBs* have multiple minimal Herbrand models. The *DDDB* consisting of a single statement, $\{a \vee b\}$, has two minimal Herbrand models, $\{\{a\}, \{b\}\}$. As shown in [Min82], to answer positive queries over *DDDBs* it is sufficient to show that the query is satisfied in every minimal model. Thus, it is not possible to answer the query, $a?$, in the above database since a is not satisfied in the model $\{b\}$. However, $a \vee b$ is satisfied in both minimal models. To answer negated queries, it is not sufficient to use the *CWA* [Rei78]: "If a cannot be proven, then assume $not\ \ a$." As noted in [Rei78], from $DB = \{a \vee b\}$, using the *CWA* we would conclude $not\ \ a$ and $not\ \ b$. But, $\{a \vee b, not\ \ a, not\ \ b\}$ is not consistent. The *generalized closed world assumption (GCWA)*, developed in [Min82] resolves this problem by specifying that the default rule for negation of an atom being considered *true* is that the atom not appear in any minimal model. This provides a model theoretic definition of negation. An equivalent proof theoretic definition ([Min82]), is that the atom a may be considered *false* if, whenever $a \vee C$ may be proven from the database, then C may be proven from the database, where C is an arbitrary positive clause.

For related work on negation in disjunctive theories see [LMR92], which also provides details of work in *disjunctive LPs (DLPs)* and *DDDBs*. See also [FM92, FM93] for a discussion of the literature on *DDDBs*.

3 Semantics for Disjunctive Deductive Databases

We consider three different classes of *DDDBs*: *DDDBs* which do not contain negated atoms in the body of rules; *disjunctive stratified databases (DSDBs)* which can contain negated atoms, but only appear in a structured way; and *disjunctive normal databases (DNDBs)* that allow the use of negated atoms in the body of rules without restrictions. In either of these cases the use of negated atoms in the body of a clause must be *safe*, i.e., any variable of a clause that occurs in a negated atom in the body must also occur in a positive atom in the body.

Since negative information is not explicitly expressed in the database, a meta rule for negation is used to derive it. For *DDDBs* negated atoms are defined by the *Generalized Closed World Assumption (GCWA)* ([Min82]) which consistently defines those atoms whose negation can be assumed to be *true* in the database. The *Extended Generalized Closed World Assumption (EGCWA)* [YH85], a generalization of the *GCWA*, is used to determine the truth or falsity of a conjunction of atoms. The negation of a conjunction of atoms is *true*, if the conjunction of the positive atoms is *false* in every minimal model.

The semantics of a *DDDB* is defined by a set of models. These models determine if a formula is a logical consequence of the *DB*. If we denote by $\mathcal{M}_{DB}$ the set of models defining the semantics of a database DB, then a positive clause C is a logical consequence of DB if and only if $\forall M \in \mathcal{M}_{DB},\ M \models C$. This characterization is called model theoretic. One of the conditions for $\mathcal{M}_{DB}$ in the different semantics to be presented, is minimality. All models in DB must be minimal models of DB. A second way to characterize the semantics of a *DDDB* is by defining $\mathcal{M}_{DB}$ as the fixpoint of a monotonic immediate consequences operator, $\mathrm{T}^{\mathrm{M}}_{DB}$ in our case.

In [FM95] an operator is described that computes the minimal models of a database DB that operates over *sets of Herbrand interpretations* in the Herbrand

Base of DB, HB_{DB}. Since our interest is on the minimal models of DB, we restrict our domain to those elements that are candidates for minimal models. The domain of our operator will be sets of minimal interpretations.

3.1 The immediate consequences operator

The family of immediate consequence operators used to describe the fixpoint semantics of different classes of programs is based on the following operator.

Definition 1 Based on [FM95]. Let DB be a *DDDB*, let $\mathcal{I}$ and $\mathcal{J}$ be a set of minimal interpretations, let S be a set of positive clauses and let I be an interpretation.

$$
\begin{array}{l}
\text{Let} \\
\qquad \mathrm{models}_I(S) = \{M \subseteq \mathrm{HB}_{DB} : \ M \text{ is a model of } (S \cup I)\} \\
\qquad \min(\mathcal{J}) = \{I \in \mathcal{J} \ : \nexists I' \in \mathcal{J}, I' \subset I\}. \\
\text{Then} \\
\qquad \mathrm{T}^{\mathrm{M}}_{DB}(\mathcal{I}) = \min(\bigcup_{I \in \mathcal{I}} \mathrm{models}_I(\mathrm{state}_{DB}(I)))
\end{array}
$$

where $\mathrm{state}_{DB}(I)$ defines the set of positive clauses which are immediate consequences of the rules in DB given that the atoms in I are considered *true*. For negation-free *DDDBs* it can be computed as follows.

$$
\mathrm{state}_{DB}(I) = \{(A_1 \vee \cdots \vee A_k)\theta : (A_1 \vee \cdots \vee A_k \leftarrow B_1, \ldots, B_n) \in DB \\
\theta \text{ is ground and } \forall i, B_i\theta \in I\},
$$

where the A_i and the B_j are atoms.

$\mathrm{T}^{\mathrm{M}}_{DB}(\mathcal{I})$ takes the set S_I of immediate consequences of the rules of DB when evaluated in I for each $I \in \mathcal{I}$ and computes the set of models of $S_I \cup I$. Then from the union of all these sets of models it selects the minimal models.

Example 1. Let $\mathcal{I} = \{\{a\}, \{b\}\}$ and let DB be a disjunctive deductive database such that $\mathrm{state}_{DB}(\{a\}) = S_1 = \{a\}$ and $\mathrm{state}_{DB}(\{b\}) = S_2 = \{a; c \vee d\}$. Then

$$
\begin{array}{l}
\mathrm{models}_{\{a\}}(S_1) = \mathcal{I}_1 = \{\{a\}, \{a,b\}, \{a,c\}, \{a,d\}, \{a,b,c\}, \ldots, \{a,b,c,d\}\} \\
\mathrm{models}_{\{b\}}(S_2) = \mathcal{I}_2 = \{\{a,b,c\}, \{a,b,d\}, \{a,b,c,d\}\} \\
\min(\mathcal{I}_1 \cup \mathcal{I}_2) = \{\{a\}\} = \mathrm{T}^{\mathrm{M}}_{DB}(\mathcal{I}).
\end{array}
$$

Under certain conditions state_{DB} is a monotonic function. When that happens, the operator $\mathrm{T}^{\mathrm{M}}_{DB}$ is also monotonic and therefore it is possible to compute its least fixpoint by computing its ordinal powers. The ordinal powers of $\mathrm{T}^{\mathrm{M}}_{DB}$ are defined in the normal way ([Llo87, LMR92]).

3.2 Semantic characterizations

We describe the semantic characterizations of *disjunctive deductive normal databases (DNDBs)*. That is, for databases with clauses of the form

$$A_1 \vee \cdots \vee A_k \leftarrow B_1, \ldots, B_n, \mathit{not}\ D_1, \ldots, \mathit{not}\ D_m \ .$$

We first present the semantic characterization of *DSDBs* and then present a semantic characterization for general *DNDBs* by introducing the use of integrity constraints. For *DNDBs*, the set of minimal models to describe its meaning does not produce the expected set of logical consequences. Let the database be, $DB = \{a \vee b \leftarrow \mathit{not}\ c\}$. This database has three minimal models $\{a\}, \{b\}$ and $\{c\}$. By the minimal models semantics, the only logical consequence of DB is the clause $a \vee b \vee c$. But the intended meaning of DB is that $a \vee b$ is a logical consequence if c is *false*, which follows since there is no rule with c on its left hand side. For the positive clause $a \vee b$ to be a logical consequence, the semantic characterization of the database must be based only on the two minimal models $\{a\}$ and $\{b\}$. For *DNDBs*, the immediate consequence operator, $\mathrm{T}^{\mathrm{M}}_{DB}$, is defined using the following definition of $\mathrm{state}_{DB}(I)$.

Definition 2 [FM95]. Let DB be a disjunctive normal database and let I be an interpretation on the Herbrand base of DB. Then

$$\begin{aligned}\mathrm{state}_{DB}(I) = \{(A_1 \vee \cdots \vee A_k)\theta : \ & \theta \text{ is a ground substitution and} \\ & (A_1 \vee \cdots \vee A_k \leftarrow B_1, \ldots, B_n, \mathit{not}\ D_1, \ldots, \mathit{not}\ D_m) \in DB \\ & \text{and } \forall i, B_i\theta \in I \text{ and } \nexists l, D_l\theta \in I\}\end{aligned}$$

where the A_i, B_j and D_l are atoms.

Disjunctive stratified databases (DSDBs). Przymusinski [Prz88] defined a model semantics for disjunctive stratified databases, called perfect model semantics. It describes the meaning of the database by a subset of the set of its minimal models. The members of this subset are called the *perfect models* (i.e. $\mathcal{M}_{DB} = \{\text{perfect Herbrand models of } DB\}$).

A *stratified disjunctive deductive database (DSDB)* is a *DNDB* that allows negated atoms in the body of a clause, but does not allow recursion through negation. In a *DSDB* it is possible to order the clauses such that they are in different strata. In a *DSDB* all clauses that do not have negation in the body of a rule may be placed in the lowest stratum. Clauses in the next stratum could contain negated atoms in their bodies provided that the atoms are defined in the previous strata. This process of developing strata continues until all clauses have been placed in strata. The stratification of the program effectively permits negated atoms to be calculated in the lower strata. A fixpoint characterization of the *perfect model semantics* using an iterative fixpoint based on the above operator, $\mathrm{T}^{\mathrm{M}}_{DB}$, and a stratification $\{DB_1, \ldots, DB_r\}$ of the clauses of DB, is defined as follows.

Definition 3. [FM95] Let DB be a stratified disjunctive database and let the set $\{DB_1, \ldots, DB_r\}$ be a stratification of the clauses of DB. Then

$$\begin{aligned}&\mathrm{T}^{\mathrm{M}}_{<DB_1>} = \mathrm{T}^{\mathrm{M}}_{DB_1} \uparrow \alpha && \text{for } \alpha \text{ a fixpoint ordinal} \\ &\mathrm{T}^{\mathrm{M}}_{<DB_1,\ldots,DB_n,DB_{n+1}>} = \mathrm{T}^{\mathrm{M}}_{DB_{n+1}} \uparrow \alpha(\mathrm{T}^{\mathrm{M}}_{<DB_1,\ldots,DB_n>}) && \text{for } \alpha \text{ a fixpoint ordinal} \\ & && \text{and } n > 0.\end{aligned}$$

Example 2. Let $DB = \{a \vee b \leftarrow \textit{not } c\}$ be a disjunctive stratified database with stratification $DB_1 = \{\}$ and $DB_2 = \{a \vee b \leftarrow \textit{not } c\}$. Then

$$\mathrm{T}^{\mathrm{M}}_{<DB_1>} = \{\emptyset\}$$
$$\mathrm{T}^{\mathrm{M}}_{<DB_1, DB_2>} = \{\{a\}, \{b\}\}.$$

In [FM95] it is shown that this iterative fixpoint generalizes the iterative fixpoint computed in the non disjunctive case by Apt, Blair and Walker [ABW88].

$$\mathrm{T}^{\mathrm{M}}_{<DB_1, \ldots, DB_n, DB_{n+1}>} = \bigcup_{I \in \mathrm{T}^{\mathrm{M}}_{<DB_1, \ldots, DB_n>}} \mathrm{T}^{\mathrm{M}}_{DB_{n+1}} \uparrow \alpha(\{I\})$$

where α is a fixpoint ordinal. $\mathrm{T}^{\mathrm{M}}_{<DB_1, \ldots, DB_r>}$ computes the *perfect models of DB* [FM95] and therefore characterizes the perfect model semantics of DB

$$\mathcal{M}_{DB} = \mathrm{T}^{\mathrm{M}}_{<DB_1, \ldots, DB_r>} \ .$$

Integrity Constraints. To introduce semantic characterizations for *DNDBs*, we consider *integrity constraints (IC)* in *DSDBs*. *ICs* describe the correct states of a database (DB). They state conditions that must be fulfilled by the theory representing the semantics of DB.

Given a model M, a ground integrity constraint, $A_1 \vee \cdots \vee A_k \Leftarrow B_1, \ldots, B_n$, is not satisfied by M if and only if $\forall i, B_i \in M$ and $\nexists j, A_j \in M$. Under the perfect model semantics, this condition can be represented by the following *denial rule*:

$$\leftarrow B_1, \ldots, B_n, \textit{not } A_1, \ldots, \textit{not } A_k$$

where this denial rule must be part of a stratum higher than those containing the rules defining the predicate symbols in the A_i and B_j. If the body of such a *denial rule* is *true* in a perfect model, that model is eliminated from consideration.

Definition 4. Given a *DSDB*, *DB*, with $\{DB_1, \ldots, DB_r\}$ as a stratification of its clauses, and with a set of *ICs* (i.e. denial rules) IC_{DB}. The following operator computes the set of perfect models of DB consistent with IC_{DB}.

$$\mathrm{T}^{\mathrm{IC}}_{DB} = \{M \in \mathrm{T}^{\mathrm{M}}_{<DB_1, \ldots, DB_r>} : M \models IC_{DB}\}$$

.

Disjunctive normal databases (DNDBs) For more general classes of *DNDBs*, which include those that allow recursion through negation, there are different competing semantics. In this paper, we use *stable model semantics* as the meaning of these databases. Stable models are a subset of the minimal models of the database and are defined using the Gelfond-Lifschitz transformation of DB.

Definition 5. [GL88] Let DB be a *DNDB* and let I be an interpretation.

$$\begin{aligned} DB^I = \{(A_1 \vee \cdots \vee A_k \leftarrow B_1, \ldots, B_n)\theta : \theta \text{ is ground and} \\ (A_1 \vee \cdots \vee A_k \leftarrow B_1, \ldots, B_n, \textit{not } D_1, \ldots, \textit{not } D_m) \in DB \\ \text{and } \{D_1\theta, \ldots, D_m\theta\} \cap I = \emptyset\}. \end{aligned}$$

DB^I is the *Gelfond-Lifschitz (GL) transformation of DB* with respect to I, where the A_i, B_j and D_l are atomic formulae.

The GL transformation produces a negation-free $DDDB$. For $DDDBs$, which usually have more than one minimal model, stable models are defined as follows.

Definition 6. [Prz90] Let DB be a disjunctive deductive normal database. M is a stable model of DB iff M is a minimal model of DB^M.

Example 3. Let $DB = \{a \leftarrow not\ b;\ b \leftarrow not\ a;\ c \leftarrow a;\ c \vee d \leftarrow b;\ d \leftarrow c\}$ be a disjunctive deductive normal database. Then the stable models of DB are: $M_1 = \{a, c, d\}$, and $M_2 = \{b, d\}$ since $M_1 \in \mathcal{M}_{DB^{M_1}}$ and $M_2 \in \mathcal{M}_{DB^{M_2}}$.

[FLMS93] show that the stable models of a *DNDB*, DB, coincide with a subset of the perfect models of a *DSDB*, $DB^{\mathcal{E}}$, that are consistent with a particular set of *ICs*. They show how to construct $DB^{\mathcal{E}}$ and the set of integrity constraints IC_{DB} from DB. This construction is performed by what is called the *evidential transformation of* DB by introducing new predicate symbols of the form "$\mathcal{E}p$" for every predicate symbol "p" in DB. Atomic formulae with these new predicate symbols are called *evidence*. Given $A \in \mathrm{HB}_{DB}$ the atom $\mathcal{E}A$ denotes an evidence for A.

Below we provide a brief description of the evidential transformation to *DNDBs*. As shown in [FLMS93], a *DNDB* characterized by the stable model semantics may be transformed, using the evidential transformation into a *DSDB*. If one computes the perfect models of the *DSDB*, and deletes the evidential atoms from each perfect model, the sets of atoms that remain are the stable models of the original *DNDB*.

First, we must introduce the concept of *semi-stratification* which will allow us to construct a stratified database $DB^{\mathcal{E}}$ based on a non-stratifiable database DB. A predicate p is said to depend upon a predicate q if there is a clause that contains p in its head and q in its body. We say p depends negatively upon predicate q if q appears negated in the body of the clause.

Definition 7. [FLMS93] Let DB be a *DNDB*. A *semi-stratification*, $\{S_1, \ldots, S_r\}$, of DB is a partition of the set of predicate symbols defined in DB such that

1. if $p \in S_i$ then any predicate q, on which p depends, belongs to a partition S_j where $j \leq i$.
2. if p depends negatively on q then $j < i$ unless q depends on p.

For *DSDBs*, stratifications and semi-stratifications coincide. In *DNDBs* all predicate symbols involved in a recursion through negation belong to the same semi-stratum, since any negative cycle in the dependency graph of the predicate symbols in the database resides in a particular strongly connected component of the graph.

Example 4. Let DB be the disjunctive normal database of Example 3. Then the following is a semi-stratification of DB

$$DB_1 = \{a \leftarrow not\ b;\ b \leftarrow not\ a\}$$
$$DB_2 = \{c \leftarrow a;\ c \vee d \leftarrow b;\ d \leftarrow c\}.$$

Given a semi-stratification, the evidential transformation of the program is defined as follows.

Definition 8. [FLMS93] Let DB be a *DNDB* with semi-stratification $\{DB_1, \ldots, DB_r\}$. The *stratified evidential transformation of DB* defines a *DSDB*, $DB^{\mathcal{E}}$, with stratification $\{DB_1^{\mathcal{E}}, \ldots, DB_r^{\mathcal{E}}\}$ and a set of *ICs*, $IC_{DB} = IC_{DB_1} \cup \cdots \cup IC_{DB_r}$, such that:

1. For each clause $A_1 \vee \cdots \vee A_k \leftarrow B_1, \ldots, B_n, not\ D_1, \ldots, not\ D_s, not\ E_1, \ldots, not\ E_m$ of DB_i, the clause $A_1 \vee \cdots \vee A_k \vee \mathcal{E}D_1 \vee \cdots \vee \mathcal{E}D_s \leftarrow B_1, \ldots, B_n, not\ E_1, \ldots, not\ E_m$ belongs to $DB_i^{\mathcal{E}}$ where the predicate symbols of the $D_l, 1 \leq l \leq s$, are defined in stratum i and the predicate symbols of the $E_j, 1 \leq j \leq m$, are defined in the strata strictly below i.
2. For each predicate symbol p defined in DB_i, the clause $\mathcal{E}p(\bar{\mathbf{x}}) \leftarrow p(\bar{\mathbf{x}})$ belongs to $DB_i^{\mathcal{E}}$.
3. For each predicate symbol p defined in DB_i, IC_{DB_i} contains an integrity constraint of the form $p(\bar{\mathbf{x}}) \Leftarrow \mathcal{E}p(\bar{\mathbf{x}})$.

Nothing else belongs either to $DB_i^{\mathcal{E}}$ or IC_{DB_i}.

Example 5. Let DB be the *DNDB* of Example 3 and DB_1 and DB_2 be the semi-stratification in Example 4. Then the evidential transformation of DB is as follows

$$\begin{aligned} DB_1^{\mathcal{E}} &= \{a \vee \mathcal{E}b;\ b \vee \mathcal{E}a; & IC_{DB_1} &= \{a \Leftarrow \mathcal{E}a;\ b \Leftarrow \mathcal{E}a\} \\ &\quad \mathcal{E}a \leftarrow a;\ \mathcal{E}b \leftarrow b\} & & \\ DB_2^{\mathcal{E}} &= \{c \leftarrow a;\ d \vee c \leftarrow b;\ d \leftarrow c; & IC_{DB_2} &= \{c \Leftarrow \mathcal{E}c;\ d \Leftarrow \mathcal{E}d\}. \\ &\quad \mathcal{E}c \leftarrow c;\ \mathcal{E}d \leftarrow d\} & & \end{aligned}$$

The perfect models of DB produced by the evidential transformation are:
$\{\{a, \mathcal{E}a, c, \mathcal{E}c, d, \mathcal{E}d\}, \{b, \mathcal{E}b, d, \mathcal{E}d\}\}$, and the stable models of DB are $\{\{a, c, d\}, \{b, d\}\}$.

4 Computing with model-trees

To exploit the fixpoint characterization of *DDDBs* in Section 3 we use the concept of a *model tree*, an abstract data structure that allows the sharing of information common to different interpretations, developed in [FM91]. A *model tree* represents sets of minimal interpretations. Each path from the root of the tree structure to a leaf node represents a minimal interpretation in the set.

Definition 9 [FM91]. Let $\mathcal{I}$ be a finite set of Herbrand interpretations (models) over a first-order language, L. *An interpretation (model) tree for* $\mathcal{I}$ is a tree structure where

- The root is labeled by the special symbol ε.
- Other nodes are labeled with atoms in $\mathcal{I}$ or by the special symbol $\not\epsilon$.
- No atom occurs twice in a branch.
- $I \in \mathcal{I}$ iff $\exists b_I\ I = \{A :\ A\ atom\ in\ b_I\}$, where b_I is a branch in the tree.

The special symbol $\not\epsilon$ is not considered an atom, it denotes the absence of an atom in the node of the tree. Its use is primarily during the answer extraction process.

4.1 Computing answers to queries

A set of answers to a DB query consists of the set of ground atoms in the DB that satisfy the query. That is, given a possibly non-ground atom, Q, representing a query, we are interested in formulas F such that $F \Rightarrow$ [4] $\exists Q$. For non-disjunctive databases, DB, the set of answers to Q can be defined by a set of ground atoms,

$$\{A \in M_{DB} : A \Rightarrow \exists Q\}$$

where M_{DB} is the unique minimal model of DB. For $DDDBs$ we must consider the fact that there may be more than one minimal model and therefore, for a formula F to be an answer, it must be *true* in every minimal model of the database. Hence the set of answers for a query can be defined as follows:

$$\{A_1 \vee \cdots \vee A_k \; : \; k > 0 \text{ and } \mathcal{M}_{DB} \models A_1 \vee \cdots \vee A_k \text{ and } \forall i, A_i \Rightarrow \exists Q\}.$$

Under this definition of answer, given a query $q(X)$, and knowing that the formulas $q(a) \vee q(b), q(a) \vee q(b) \vee q(c)$ and $q(a) \vee q(b) \vee p(c)$ are logical consequences of the database, we can say that the first two formulas constitute answers to the query. The third formula, $q(a) \vee q(b) \vee p(c)$, does not constitute an answer because $p(c)$ does not imply $\exists X q(X)$. If we are interested in a complete set of answers, then $q(a) \vee q(b)$ is sufficient to represent all answers, since it subsumes $q(a) \vee q(b) \vee p(c)$.

We want to extend the evaluation method of definite databases to disjunctive databases. To find an answer to a query in a $DDDB$ in its set of minimal models is equivalent to finding in each model an atom that implies the query and then constructing a disjunction with these atoms.

Definition 10 [FM91]. Let DB be a DDDB and let Q be a query. $\mathcal{M}_{DB}[Q]$ is the *set of minimal support of* Q *with respect to* DB.

$$\mathcal{M}_{DB}[Q] = \min(\{M[Q] : M \in \mathcal{M}_{DB}\})$$

where $M[Q] = \{A \in M : A \Rightarrow \exists Q\}$. $\mathcal{M}_{DB}[Q]$ is called the *set of minimal models for* answers *to the* query Q *in* DB.

If a positive query, Q, is not *true* in a model, M, then $M[Q]$ is empty.

Theorem 11 [FM91]. *Let DB be a DDDB and let Q be a query, then*

1. $\mathcal{M}_{DB}[Q] = \{\emptyset\}$ *iff* $DB \not\models \exists Q$.
2. C *is an answer to* Q *iff* $\mathcal{M}_{DB}[Q] \models C$.

We can use Theorem 11 to devise an algorithm to compute a tree corresponding to the set of minimal support for a query Q with respect to a database DB. We call this tree *the answer tree for* Q *in* DB, $\mathcal{T}_{DB}[Q]$. The algorithm computes a model tree for the database DB and then applies Theorem 11 to extract the answer tree.

As noted in Definition 9, the symbol $\not\varepsilon$ is not considered to be an atom and therefore its occurrences can be eliminated during Step 5 of Algorithm 1.

Theorem 12 [FM91]. *Let DB be a DDDB and let Q be a query, then the application of Algorithm 1 to the model tree $\mathcal{T}_{DB}$ and the query Q produces the answer tree $\mathcal{T}_{DB}[Q]$ that represents the set of interpretations $\mathcal{M}_{DB}[Q]$.*

[4] $\Rightarrow$ denotes classical implication

Algorithm 1. *Computing an answer tree for a query.*
Let $\mathcal{T}_\mathcal{I}$ be an interpretation tree for a set of minimal interpretations $\mathcal{I}$ and let Q be a query.

1. $T \leftarrow \varepsilon$.
2. Let $\mathcal{D}_\mathcal{I}$ be the atoms occurring in $\mathcal{T}_\mathcal{I}$.
3. Let $\mathcal{A} = \mathcal{D}_\mathcal{I}[Q]$.
4. For each branch b of $\mathcal{T}_\mathcal{I}$
   Let $\mathcal{A}_b = \{A \in \mathcal{A} : A \text{ occurs in } b\}$.
   If $\mathcal{A}_b = \emptyset$ then Return ε.
   Let $\mathcal{A}_b = \{A_1, \ldots, A_{n_b}\}$ such that $i < j$ iff A_i is an ancestor of A_j in branch b of $\mathcal{T}_\mathcal{I}$.
   Add to T a new branch that corresponds to the path $(\varepsilon, A_1, \ldots, A_{n_b}, \not\varepsilon)$.
5. Eliminate any non minimal branch of T.
6. Return T.

5 Updating Disjunctive Deductive Databases

The view update problem is that of performing an update of an *IDB* predicate by modifying the underlying relations in the *EDB* part of the database. *EDB* and *IDB* predicate names are assumed to be distinct. In the case of purely definite *DDDBs*, where the update must maintain the definiteness of the *DB*, it is unclear as to how to modify the *EDB* relations to accomplish some view updates. Consider the simple database

$$P(x) \leftarrow A(x)$$
$$P(x) \leftarrow B(x),$$

where P is an intensional predicate, and A and B are extensional. There are only three plausible ways to update this *DB* with the information $P(c)$, when we are restricted to definite clauses: add $A(c)$, add $B(c)$, or add both $A(c)$ and $B(c)$. Each of the first two options seems arbitrary; the third results in an update that is too strong. Permitting disjunctive information, the update can be accomplished by adding $A(c) \vee B(c)$; and this option permits a conservative approach, sometimes called *skeptical* to updating the database without committing to either $A(c)$ or $B(c)$.

There have been several papers on updating deductive databases. [FUV83] provide a semantic characterization of correctness of updates in *DDBs* that forms the basis of the work of [GHLM93], but do not provide update algorithms. See [GHLM93, FGM94] for related literature.

We follow the work of [GHLM93] in discussing the semantics of view updates in *DDDBs*. The view update insertion algorithm assumes that there is an algorithm to insert and delete disjunctions of atoms from the *EDB*, as presented in [GGM95]. Since we are limited to modifying only the *EDB*, some updates will not be possible.

For algorithms on deletion of view predicates in *DDDBs* and *DSDBs*, see [GHLM93], and for *DNDBs* see [FGM94].

We assume the *IDB* is definite, i.e., it consists of Horn clauses, while the *EDB* is disjunctive. Horn clauses in the IDB are assumed to be fully ground. This avoids the problem of possibly having to deal with null values in inserts. Alternatively one may assume that all variables in the body occur in the head. Both assumptions avoid the null value problem. The algorithm for insertion to be described below is given in terms of restricted SLD-trees [Llo87]. A restricted SLD-tree is defined as follows.

Definition 13 Based on [Llo87]. Let P be a definite program and G a definite goal. Let R be a computation rule that only selects intensional atoms. A *restricted SLD-tree* for $P \bigcup G$ is a tree satisfying the following:

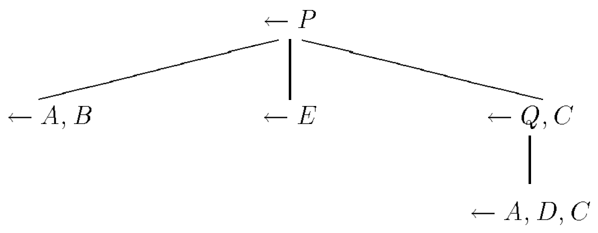
- Each node of the tree is a (possibly empty) definite goal.
- The root node is G.
- Let $\leftarrow A_1, \ldots, A_m, \ldots, A_k (k \geq 1)$ be a node in a tree and suppose that A_m is the atom selected by the computation rule R. Then, for each input clause $A \leftarrow B_1, \ldots, B_q$ such that A_m and A are unifiable with most general unifier θ, the node has a child $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$
- Nodes which are empty or consist only of extensional atoms, have no children.

An example of a restricted SLD-tree that finds all proof trees is:

Example 6. Let *DB* be the disjunctive database:

$$\begin{array}{ll} P \leftarrow A, B & \qquad A \vee B \vee C \\ P \leftarrow E & \\ P \leftarrow Q, C & \\ Q \leftarrow A, D & \end{array}$$

Let $\{A,\ B,\ C,\ D,\ E\}$ be the set of *EDB* atoms and $\{P,\ Q\}$ be the set of *IDB* atoms. The restricted SLD-tree for the goal $\leftarrow P$ in *DB* is:



Example 6 finds all restricted SLD-trees for a goal. Each branch terminates with a goal clause that consists exclusively of *EDB* atoms. We define a *flat clause* to be one formed by the disjunction of the conjunction of atoms in each terminal goal node of the restricted SLD-tree that finds all derivations. P is a logical consequence of DB if the negation of any flat clause associated with the leaves in a restricted SLD-tree for the goal $G = (\leftarrow P)$ is *true* in DB. For example in the leftmost leaf in Example 6, if A and B are true in DB, then P is also true in DB. Since it is desirable to "minimize" the changes to DB, the insertion algorithm considers the weakest formula that achieves the insertion. In the example, this formula is given by the disjunction of the leaves: $(A \wedge B) \vee (E) \vee (A \wedge D \wedge C)$.

Definition 14. Let DB be a $DDDB$. An insertion of an atom P is a *possible insertion* into DB iff there exists a $DDDB$ DB' such that $I_{DB'} = I_{DB}$ and $DB' \vdash P$.

Algorithm 2. [GHLM93] *Insertion of an intensional atom P into a disjunctive database DB.*
Given an atom P and a disjunctive database DB such that $DB \nvdash P$, the algorithm computes a database DB' such that $DB' \vdash P$ whenever the insertion is possible.

1. Let $DB' = \emptyset$.
2. Construct a restricted SLD-tree for $(\leftarrow P)$ from DB.
3. Let $F_1, \ldots, F_n$ be the negations of flat clauses associated with leaf nodes containing only conjunctions of extensional atoms. If $n = 0$ the insertion fails. Otherwise, construct the conjunctive normal form of $F_1 \vee \cdots \vee F_n$, writing it as $C_1 \wedge \cdots \wedge C_m$, where subsumed clauses are omitted from the conjunctive normal form.
4. For each C_i, $1 \leq i \leq m$, if $DB \nvdash C_i$ (i.e. there is no disjunction in the extensional part of DB that subsumes C_i) then insert C_i in DB'.
5. Add to DB' all the clauses in DB.

[GHLM93] show that Algorithm 2 accomplishes the desired insertion.

Example 7. From the restricted SLD proof of Example 6 we obtain $F_1 = A \wedge B$, $F_2 = E$ and $F_3 = A \wedge D \wedge C$. Then, $DB \vdash P$ if $DB \vdash F_1 \vee F_2 \vee F_3$. The conjunctive normal form of $F_1 \vee F_2 \vee F_3$ is $C_1 \wedge C_2 \wedge C_3$, where $C_1 = A \vee E$, $C_2 = B \vee D \vee E$, $C_3 = B \vee C \vee E$. Then, the updated database DB' is:

$$\begin{array}{ll} P \leftarrow A, B & A \vee B \vee C \\ P \leftarrow E & A \vee E \\ P \leftarrow Q, C & B \vee C \vee E \\ Q \leftarrow A, D & B \vee D \vee E \end{array}$$

To accomplish the insertion, [GHLM93] require that the insertion modifies the original database as little as possible, where this is defined as follows.

Definition 15. Let L be a function-free first order language. Let DB be a disjunctive database in L. Let P be an intensional atom and assume $DB \not\vdash P$. A *minimal insertion* of P into DB is a minimal superset DB' of DB that implies P. That is, there is no database DB'' such that $DB \subseteq DB'' \subset DB'$, and $DB'' \vdash P$.

As shown in [GHLM93], Algorithm 2 produces a minimal insertion.

The minimality condition alone does not imply uniqueness, and so it cannot be used as a criterion to justify the insertion algorithm. To force a unique result, [GHLM93] requires not only that the update be minimal, but that it yield the weakest modification of the database, in the following sense.

Definition 16. Let DB_1 and DB_2 be disjunctive databases. DB_1 is *weaker* than DB_2 iff $DB_2 \vdash DB_1$. Let $\mathcal{DB}$ be a set of disjunctive databases. DB is the *weakest* disjunctive database in $\mathcal{DB}$ iff for any other disjunctive database DB' in $\mathcal{DB}$, DB is weaker than DB'.

It is shown in [GHLM93] that Algorithm 2 constructs the weakest of the minimal disjunctive databases that accomplish the desired update.

6 Proposed Architecture for a DDDB

In the context of *DDDBs* we have described algorithms based upon *model trees* to compute answers to queries in *DDDBs*. Now, given a *DDDB*, consisting of a set of clauses, it is possible to cluster the clauses such that clauses that share atoms are in the same cluster. One can then have model trees for each cluster. We call the set of model trees a *model forest* ([FM92, FM93]. For a query that has atoms that appear in only one model tree it is only necessary to determine if the query is satisfied by that model tree. If it is, then one need not look at any other model tree to determine the validity of the query. The number of models in a clustered *DDDB* consists of the product of the number of models in each model tree in the model forest. Hence, restricting the search to one, or a small number of model trees is important for efficiency considerations.

In developing an architecture for a *DDDB*, one wants to take advantage of the large amount of work that has been accomplished on relational and *DDBs*. A *DDDB* will have several different kinds of predicates. There will be predicates that are exclusively relational, i.e., they are not defined by rules. There are predicates that are defined by Horn rules. There are also extensional predicates that appear in ground disjunctions, and predicates that are disjunctive in that they are defined by rules that are disjunctive, or have predicates in the body of the rules that are disjunctive. In any *DDDB* it is straightforward to specify the class to which predicates belong, as well as to specify which predicates belong to which stratum of a *DSDB*. Now, given a query, it is possible to determine to which class the predicate belongs. If all predicates belong to a relational database, then the query should be handled in the same way as in such a database and with the same complexity. If the query has all predicates that are deductive, but not disjunctive, then it should be handled by a *DDB*. If it is purely disjunctive, then it should be handled by a *DDDB*, that operates upon the model trees. If a query has predicates that belong to the different classes, then the different processors can interact with one another.

In the architecture sketched below, the complexity related to answering a query arises when we are in the disjunctive processor. The time to respond to a query depends upon the number of model trees in the model forest that must be accessed and the number of models in the individual trees. In Section 6.1 we describe the proposed architecture. In Section 6.2 we provide a simple model to indicate the class of *DDDBs* that are "effective" using the proposed architecture.

6.1 DDDB Architecture

We now put these results together and present, in Figure 1, an architecture for a disjunctive deductive database management system, proposed by Fernández [Fer94].

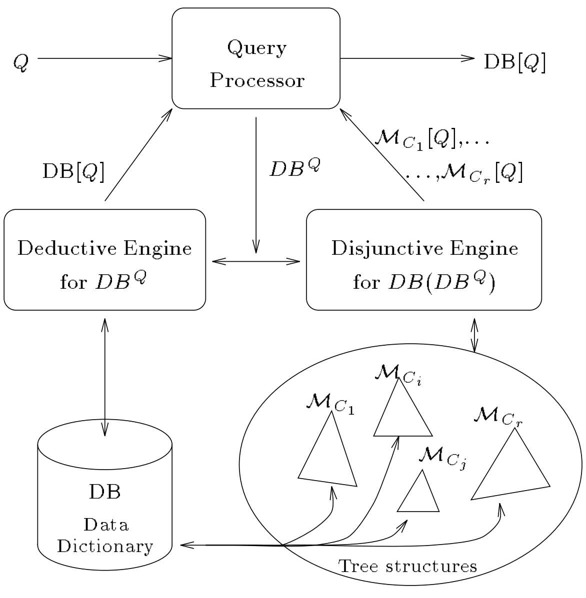


Fig. 1. Architecture for a Disjunctive Deductive Database System

In this architecture, the evaluation of a query, Q, in a disjunctive database DB is performed by a *query processor* that selects the relevant clauses of DB with respect to the query, DB^Q, and evaluates each of its clauses in a process that is divided into two parts:

1. a *deductive part*, consisting of a *deductive engine* that computes the minimal model of the Horn-database, DB^Q, over the data dictionary. The data dictionary consists of the ground predicates in the DB and should contain pointers to where the predicates are stored. As a byproduct, it generates ground substitution, $\theta_1, \ldots, \theta_l$, for disjunctive clauses; and
2. a *disjunctive part*, consisting of a *disjunctive engine* that takes the ground instances generated by the deductive engine and manipulates tree structures,

$\mathcal{M}_{C_1}, \ldots, \mathcal{M}_{C_r}$, that correspond to the clustered representation of the indefinite information in the models of the disjunctive database.

The query processor collects the answers and builds the corresponding answer forest, $DB[Q]$. Notice that when a query is definite, its evaluation involves only definite predicates and therefore no disjunctive clause has to be evaluated. This means that no instances of disjunctive clauses are generated and therefore there is no participation of the disjunctive engine in the computation of the answers. Hence the architecture reduces to a deductive or relational database system in these cases. When the query involves indefinite information, the cost of its processing depends on the interaction between tree structures during the evaluation of the query. The more interaction (i.e. need to combine tree structures) between the clusters, the more complex the processing performed by the disjunctive engine.

The deductive and disjunctive engines, can operate with a high degree of independence. This means that on a concrete implementation of this architecture one will be able to take advantage of the future developments in the area of *DDB* technology when designing the implementation of the deductive engine. Moreover, one can conceive that for the deductive engine it would be possible to use an "off the shelf" *DDB* management system. The design of a concrete implementation must concentrate on the problem of storage and management of the tree structures.

6.2 Complexity Considerations

In general, the complexity of answering queries in *DDDBs* is exponential in the number of disjunctive clauses. However, as noted above, not all queries require exponential time to be answered. Queries that are completely relational or deductive will be answered in times that depend upon the size of the database. With the architecture we have described, mixed queries or completely disjunctive or deductive disjunctive may be exponential. But, even in these cases, the complexity may be reasonable. This arises when disjunctive predicates are in the same tree and the number of models in the model tree are relatively small. The complexity is high when the model tree has a large number of minimal models or the conjunction of predicates are from different model trees and each model tree is relatively large.

In the following analysis we make several simplifying assumptions to provide some insight as to the size of the model trees that can exist and yet have an "effective" computation. By "effective" we mean that the time to respond to a conjunction is no more than twice the time that it would take if all predicates in the conjunction were relational. We do not consider the possibility of optimization of a query in either the definite or indefinite part of the query. We assume a join requires comparing every tuple in a relation with every tuple in the other relations involved in a query. Similarly, we assume that for indefinite answers we need to access every node of the model tree (several times if any bindings are passed from the definite part of the query to the indefinite part). This provides a worst case analysis. We believe that it is possible to optimize search over the disjunctive part of the database, just as there is for the definite part [Ull88a, Ull88b], but do not address this topic here.

We denote a query by:

$$\text{Query: } \leftarrow p_1, \ \ldots, p_l, \ q_1, \ \ldots, q_m,$$

where the predicates p_i are relational and the predicates q_j are disjunctive. All relational predicates are assumed to have the same number of tuples and that the

$$\text{Number of tuples in a relational table} = \text{T}.$$

For the disjunctive part we assume that each cluster (model tree) has the same size, given by:

$$\text{Number of disjunctions per cluster} = \text{D},$$

and the size of each disjunction, i.e., the number of disjuncts in each disjunction is:

$$\text{Size of disjunctions} = \text{S}.$$

If the p's and q's do not share variables there are no bindings (i.e, substitutions for the variables in the p's and in the q's) to be passed from the definite into the indefinite part of the query. Thus, we need to access every node of the tree only once. Since there are m disjunctive predicates in the query, we need (in the worst case), to merge m different clusters. Hence, the number of disjunctions represented in the merged tree is: $m * D$. The number of branches is equal to: S^{m*D}, which can be approximated as the number of nodes in the tree (since the latter is, for balanced trees, at most twice as big as the former). Thus we get:

$$\text{Complexity of answering a query} = \; S^{m*D} \; + \; T^l$$

If the p's and q's share variables, each node has to be accessed as many times as there are bindings passed from the definite part of the query. Thus the above formula becomes:

$$\text{Complexity of answering a query} = \; B * S^{m*D} \; + \; T^l,$$

where B is the number of bindings passed from the definite to the indefinite part of a query.

With the above estimates of complexity we can compute the respective sizes of tree clusters and relational tables for various parameters. The following two graphs show (for $m, l = 2$) points at which the two parts of the equation are equal, i.e. when the complexity of the definite part is equal to the complexity of the indefinite part.

7 Conclusions

We have provided a review of some semantics for *DDDBs*. In particular, we described the semantics of *DDDBs* that do not have negation by default in their clauses, *DSDBs* that have negation in the body of rules, but do not have recursion through negation, and *DNDBs* that may have recursion through negation, but comply with the stable model semantics. In each case we described how to represent and manipulate the database using the concept of *model trees*. We also described how the view update problem can be solved in a *DDDB*, and referred to work that handles the view update problem in *DSDBs* and *DNDBs*.

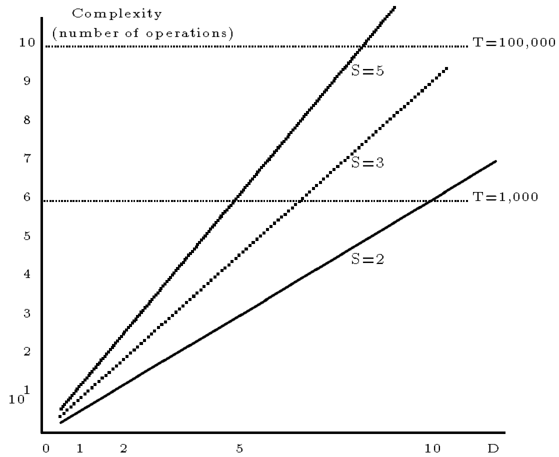


Fig. 2. Comparison of complexity: definite vs. indefinite part of the query (p's and q's do not share variables). As shown in this figure, processing a join over two 1,000 tuple tables, takes about as much time as retrieving answers from a cluster consisting of 10 disjunctions of size 2, or 5 disjunctions of size 5.

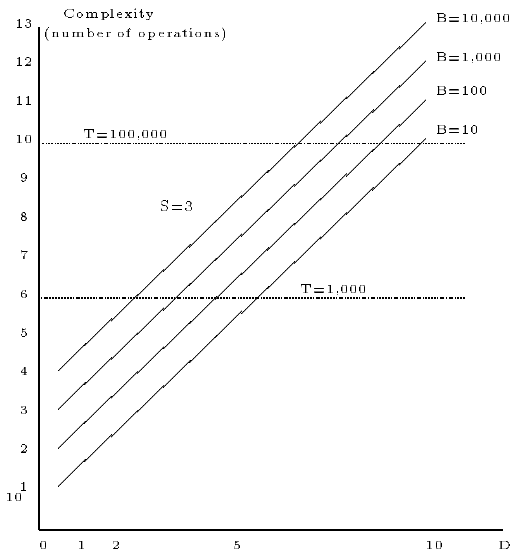


Fig. 3. Comparison of complexity: definite vs. indefinite part of the (p's and q's share variables). Here the situation is somewhat more complex: a 2-way join over a table of size 1,000 is as expensive as processing answers in cluster with 5 disjunctions of size 3 when 10 alternative bindings are passed form the definite part, or a cluster of only 3 disjunctions when 1,000 (i.e. the entire table) is passed as bindings.

We then described an architecture for *DDDBs* that can handle any of the three classes of disjunctive databases discussed in this paper. The architecture takes advantage of relational and deductive technologies. We describe a class of disjunctive databases that will be "effective". Although disjunctive databases may have a large number of disjunctions, it is doubtful that all of the disjunctions will appear in the same cluster. Hence, it is likely that most queries will be answered in times that are not excessive, and are "effective". Since the time needed to answer a query is a function of the number of model trees involved in the query and the number of models in each tree, there may be some queries for which it will take considerable time to find an answer. In these cases it will be possible to inform the user when such queries arise. Users who need the answers and who wish the query to be answered can do so, while others may choose to terminate the query. A prototype disjunctive database that consists only of positive clauses without bodies has been implemented and may be accessed through the World Wide Web (http:karna.cs.umd.edu:3264/projects/dddbs/dddbs_main.html).

Based upon the above analysis, there is reason to believe that disjunctive databases will become practical tools. As many theories in non-monotonic reasoning can be represented as *DDDBs*, such theories will be able to take advantage of *DDDB* technology ([Min93]).

Acknowledgements

This work was supported by the National Science Foundation under grant number IRI-9300691. We also express our appreciation to Parke Godfrey and Carolina Ruiz for their assistance with the paper.

References

[ABW88] K. Apt, H. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann Pub., Washington, D.C., 1988.

[Fer94] J. A. Fernández. *Disjunctive Deductive Databases.* PhD thesis, University of Maryland, Department of Computer Science, College Park, 1994.

[FGM94] J. A. Fernández, J. Grant, and J. Minker. Model theoretic approach to view updates in deductive databases. Technical Report CS–TR–3335, Department of Computer Science, University of Maryland, College Park, MD 20742, 1994.

[FLMS93] J. A. Fernández, J. Lobo, J. Minker, and V. Subrahmanian. Disjunctive LP + integrity constraints = stable model semantics. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):449–474, 1993.

[FM91] J. A. Fernández and J. Minker. Bottom-up evaluation of Hierarchical Disjunctive Deductive Databases. In K. Furukawa, editor, *Logic Programming Proceedings of the Eighth International Conference*, pages 660–675. MIT Press, 1991.

[FM92] J. A. Fernández and J. Minker. Semantics of disjunctive deductive databases. In *International Conference on Database Theory: Lecture Notes in Computer Science 646*, pages 21–50. Springer-Verlag, 1992. Invited Paper.

[FM93] J. Fernández and J. Minker. Theory and algorithms for disjunctive deductive databases. *Programmirovanie*, N 3:5–39, 1993. (also appears as University of Maryland Technical Report,CS-TR-3223, UMIACS-TR-94-17,1994. Invited Paper in Russian).

[FM95] J. Fernández and J. Minker. Bottom-up computation of perfect models for disjunctive theories. *Journal of Logic Programming*, 1995. To appear.

[FUV83] R. Fagin, J. Ullman, and M. Vardi. On the semantics of updates in databases. In *Proc. 7th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 352–365, 1983.

[GGM95] J. Grant, J. Gryz, and J. Minker. Updating disjunctive databases via model trees. Technical Report CS-TR-3407 and UMIACS-TR-95-11, Department of Computer Science and the University of Maryland Institute for Advanced Computer Studies, College Park, Maryland 20742, 1995.

[GHLM93] J. Grant, J. Horty, J. Lobo, and J. Minker. View updates in stratified disjunctive databases. *Journal of Automated Reasoning*, 11:249–267, 1993.

[GL88] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In R. Kowalski and K. Bowen, editors, *Proc. 5th International Conference and Symposium on Logic Programming*, pages 1070–1080, Seattle, Washington, August 15-19 1988.

[Llo87] J. Lloyd. *Foundations of Logic Programming.* Springer–Verlag, second edition, 1987.

[LMR92] J. Lobo, J. Minker, and A. Rajasekar. *Foundations of Disjunctive Logic Programming.* MIT Press, 1992.

[Min82] J. Minker. On indefinite databases and the closed world assumption. In *Lecture Notes in Computer Science 138*, pages 292–308. Springer-Verlag, 1982.

[Min86] J. Minker, editor. *Proceedings of Workshop on Foundations of Deductive Databases and Logic Programming*, August 1986.

[Min88] J. Minker. Perspectives in deductive databases. *Journal of Logic Programming*, 5:33–60, 1988.

[Min89] J. Minker. Toward a foundation of disjunctive logic programming. In *Proceedings of the North American Conference on Logic Programming*, pages 121–125. MIT Press, 1989. Invited Banquet Address.

[Min93] J. Minker. An overview of nonmonotonic reasoning and logic programming. *Journal of Logic Programming*, 17(2, 3 and 4):95–126, November 1993.

[Prz88] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programming. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann Pub., Washington, D.C., 1988.

[Prz90] T. C. Przymusinski. Extended stable semantics for normal and disjunctive programs. In D. Warren and P. Szeredi, editors, *Proceedings of the 7th International Logic Programming Conference*, pages 459–477, Jerusalem, 1990. MIT Press. Extended Abstract.

[Rei78] R. Reiter. On Closed World Data Bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, New York, 1978.

[Ull88a] J. D. Ullman. *Principles of Database and Knowledge-Base Systems I.* Principles of Computer Science Series. Computer Science Press, Rockville, Maryland 20850, 1988.

[Ull88b] J. D. Ullman. *Principles of Database and Knowledge-Base Systems II.* Principles of Computer Science Series. Computer Science Press, Rockville, Maryland 20850, 1988.

[YH85] A. Yahya and L. Henschen. Deduction in Non-Horn Databases. *J. Automated Reasoning*, 1(2):141–160, 1985.