An overview of deductive databases with an emphasis on problems related to negative information is presented. The subject is embedded into the wider context of logic programming, exposing certain peculiarities pertinent to treatment of negation in this field.

Introduction

In this paper, we discuss problems related to representing and handling negative information in deductive databases. The subject, albeit over ten years old, has not lost anything from its appeal and momentum. To the contrary, it appears to gain continuously even more attention from the database community. It may be embedded in the context of another, nowadays fashionable topic known under the name of non-monotonic logic.

The deductive model of a database emerged from its predecessor, the relational model, by incorporating certain elements and methods of automated theorem proving. In the relational model of a database facts are organized in a finite collection of relations with the standard operations on these relations such as: insertion, deletion, selection, projection, product, join, union, intersection, and difference. Every relation is represented by a finite set of tuples. Database users access the information stored in the database by means of queries expressed in a suitable language. The database management system plays the role of an interface between the user and the physical database, and is responsible for accepting the user’s queries and operations as well as returning answers to the user. Moreover, it ensures that the integrity constraints pertinent to the database are not violated. Although this last task involves certain elements of reasoning, because the integrity constraints are usually expressed in the form of rules, the relational database, or rather its management system, does not go much beyond passive verification of the legality of the performed operations against the coded integrity constraints. Another feature of relational databases that requires some form of reasoning is the concept of view, which may be interpreted as rule based definitions of relations not explicitly represented in a physical database. However, views are usually limited to certain non-recursive rules and are intended to hide information rather than to equip a database with deductive abilities.

The seemingly natural observation, that an active use of rules which are present in a database may result in the derivation of new facts, not explicitly represented in it, gave rise to deductive databases. Conceptually, a deductive database may be seen as a relational database furnished with additional, “intelligent” layers capable of complex reasoning from the content of the relational component. In fact, this point of view seems to prevail. One should note, however, that since contemporary rela-
tional databases are not purely passive in executing their integrity constraints, the distinction between relational and deductive databases is rather fuzzy, and one certainly can find a variety of cases which may be classified “relational” as well as “deductive”. On the other hand, their different forms of representation require, for the purpose of making a deductive database out of a relational one, a suitable translation process. It turns out that this process is largely responsible for the need for non-standard treatment of negation in processing queries against deductive databases.

If one translates the content of a purely relational database into its deductive counterpart, the problem with the interpretation of negative information manifests itself. Certainly, while translating tuples onto predicates, or more precisely, onto ground literals, only the positive information of the relational part is explicitly processed, despite the fact, that the relational part, by the absence of some tuples, may encode negative information as well. The first systematic treatment of this phenomenon by means of closed world assumption (cwa) may be found in (Reiter, 1978b), and since then, a mature age of deductive databases seems to have begun.

The closed world assumption may be recognized as a benchmark which makes a difference between relational and deductive databases, since in the relational model the closed world assumption is somewhat hardwareed, and is therefore superfluous, while deductive databases do need cwa to allow negative conclusions from purely positive information. Under cwa, all simple positive facts that cannot be derived from the database are assumed to be false, or in other words, their negations are asserted. This straightforward and seemingly unquestionable rule creates serious problems, however, when mechanically applied to deductive databases which are not the result of translating the content of a relational database. That is why a proper treatment of negation is so crucial in this context.

In the present understanding of the term, deductive databases consist of the following syntactic elements:
1. Extensional part, which corresponds to the relations, and usually has the form of a set of so called ground positive literals,
2. Intensional part, which corresponds to view definitions, integrity constraints, and other dependencies, and usually has the form of a set of so called clauses,
3. Resolution theorem prover, which derives logical consequences of the extensional and intensional content of the database, and
4. Meta-rules for negative inferences, e.g., the closed world assumption.

In this paper, after a general introduction to deductive databases, we focus on the last element of this list, discussing how they affect the entire database and its information content, and what meaning may be consistently associated with such rules. Because the area of deductive databases is not the only one concerned with proper treatment of negation, we also briefly discuss how the problem is approached in logic programming. We conclude with a succinct history of developments in this area, mentioning current experimental research.

Preliminaries

Deductive databases have the form of a collection of clauses expressible in a language of first-order logic. In this section, we provide a concise introduction to first-order clause logic, and its minimal model semantics.

Syntax of First-Order Logic. Logic is a branch of mathematics which investigates infallible rules of reasoning. Propositional logic is, perhaps, the simplest form of logic which deals with propositional statements, such as “It is sunny today” or “It is not raining today”, and reasoning within this context. Propositional symbols together with the falsehood symbol □ are used to represent statements, and may incorporate logical connectives ∧ (and), ∨ (or), → (implies), and ¬ (not) to combine simple statements into more complex ones. For example, if P denotes “It is sunny today”, and Q denotes “It is not raining today”, then P → Q denotes “If it is sunny today, then it is not raining today”. Other customarily used connectives may be understood as abbreviations. In particular, the equivalence P — Q stands for the two way implication (P → Q) ∧ (Q → P).

Despite its elegant simplicity, propositional logic is neither particularly suitable to proper treatment of statements of the form: “All men are mortal”, “Socrates is a man”, nor to reason within this context. For example, it is not possible to conclude that “Socrates is mortal” from these statements using only propositional axioms and modus ponens. This kind of limitation of propositional provability does not appear in first-order logic, which is a more powerful form of logic capable of formally expressing such statements and reasoning with them. In addition to logical connectives, first-order logic allows quantifiers ∀ (for all) and ∃ (there exists) ranging over individual variables appearing in statements. Moreover, it contains more axioms and an extra rule of inference. For example, given (∀x) (Man(x) → Mortal(x)), which formally expresses the statement “All men are mortal”, and Man(Socrates), which articulates the statement “Socrates is a man”, one can conclude Mortal(Socrates), which denotes “Socrates is mortal”.

Since part of first-order logic provides a framework for deductive databases, we will take a closer look at
its syntax and semantics. The alphabet for first-order language consists of predicate symbols, constant symbols, variable symbols, function symbols, quantifiers, logical connectives, and other usual punctuation symbols. For simplicity, we follow a general trend in deductive databases and logic programming not to include the equality symbol (=) in this language. There are essentially two different kinds of expressions one can build out of the symbols of this alphabet: terms and formulas. Terms provide a repertoire of names for objects of the universe of discourse, like “Socrates”, “x”, “7”, “f(x,g(7))”, etc. More formally, a term is a constant symbol, or a variable symbol, or a function symbol followed by a list of arguments which are terms themselves, with punctuation symbols used in a usual way. In the above examples “Socrates” and “7” are constants, “x” is a variable, and “f(x,g(7))” is a complex term, where f and g are functions. Formulas are used to express attributes and relationships pertinent to these objects. The statement attributing humanity to Socrates, “Socrates is a man”, is formally expressed by Man(Socrates) using a unary predicate Man and the constant Socrates. The statement relating Tom and Mary, “Tom likes Mary”, is expressed by Likes(Tom,Mary) using a binary predicate Likes and the constants Tom and Mary. One can also use functions to represent statements. For example, the statement “Tom resents Mary’s father” may be articulated as Respects(Tom,father(Mary)). A mathematical statement which states that x is less than f(x,g(7)) can be written as “< (x,f(x,g(7)))”, or in a more familiar infix form, as “x < f(x,g(7))”.

Simple statements of the above form are called atomic formulas. More formally, an atomic formula is a predicate symbol, followed by a list of arguments, which are terms separated by punctuation symbols. Quantifiers and logical connectives allow, in an obvious way, composition of simpler formulas into more complex ones. Atomic formulas and negated atomic formulas are referred to as positive literals and negative literals respectively. The term literal denotes both positive and negative literals.

Formulas expressible in first-order language may be fairly complicated as, for instance, the following one:

\[(\forall x)(\exists y)(\forall z)(\text{Belongs-To}(z,y) \equiv (\exists v)(\text{Belongs-To}(v,x) \land \text{Belongs-To}(z,v)))\]

It should not be surprising that handling complex formulas like this in a deductive database, if at all feasible, may be very costly and time consuming. Fortunately, although literals do not seem to have enough expressive power, such complex expressions as the one above are hardly ever encountered in database applications. It has been widely accepted that the class of clauses, which is a proper subclass of all first-order formulas, is satisfactorily sufficient for this purpose.

The notion of clause is a key syntactic concept in deductive databases. Clauses allow for expressing statements more sophisticated than literals, while maintaining relative simplicity so desired in prospective applications of deductive databases. Using them, one can represent dependencies and constraints of various kinds.

Let us take a look at two examples of statements one may wish to represent in a deductive database: “for every x and y, if x is the parent of y then x is either the father of y or the mother of y”, and “for every x, y, and z, if x is a component of y and y is a component of z, then x is a subcomponent of z”. The first statement can be articulated, using the predicates Father, Mother and Parent as:

\[(\forall x)(\forall y)(\text{Parent}(x,y) \rightarrow \text{Father}(x,y) \lor \text{Mother}(x,y))\]

and the second one by using the predicates Component and Subcomponent as:

\[(\forall x)(\forall y)(\forall z)(\text{Component}(x,y) \land \text{Component}(y,z) \rightarrow \text{Subcomponent}(x,z))\]

In the context of deductive databases, the above formulas are written in a reversed direction without quantifiers as:

\[\text{Father}(x,y) \lor \text{Mother}(x,y) \rightarrow \text{Parent}(x,y)\]

\[\text{Subcomponent}(x,z) \rightarrow \text{Component}(x,y) \land \text{Component}(y,z)\]

More generally, a clause is a formula of the form:

\[P_1 \lor \ldots \lor P_m \leftarrow Q_1 \land \ldots \land Q_n\]

where \(P_i\)s and \(Q_j\)s are positive literals.

A clause is said to be a definite clause if \(m = 1\) and an indefinite clause (sometimes referred to as disjunctive clause) if \(m > 1\). For example, Grandfather(x,y) \leftarrow Father(x,z) \land Father(z,y) is a definite clause and Father(x,y) \lor Mother(x,y) \leftarrow Parent(x,y) is an indefinite clause. The left hand side of the clause is referred to as the head and the right hand side as the body of the clause. If the body of a clause is empty, the \(\leftarrow\) symbol is dropped for notational convenience. A ground clause is a clause with no occurrences of variables. They involve neither explicit nor implicit quantifiers, and therefore are essentially equivalent in their expressive power to sentences of propositional logic. A positive clause is one with an empty body. A negative clause is one with an empty head; it is logically equivalent to the negation of its body (i.e., one proves another and vice versa). An empty clause consists of no literals and represents a contradiction; it is denoted by the falsehood symbol \(\Box\). Figure 1 summarizes the various kinds of clauses.

Although clauses constitute a small subclass of all first-order formulas, their expressive power is surprisingly strong. For instance, every sentence of the form:

\[\forall x_1 \ldots \forall x_n \Phi(x_1, \ldots, x_n)\]

where \(\Phi(x_1, \ldots, x_n)\) is an arbitrarily complex quantifier-free formula with variables \(x_1, \ldots, x_n\), is logically equivalent to a finite set of clauses. Also, existential statements may be approximated using auxiliary constants or function symbols. These facts explain, to some extent, why the language of clauses has been widely
accepted for deductive databases, logic programming, and other domains of artificial intelligence.

Semantics of First-Order Logic. First-order clauses provide deductive databases not only with form, but also with meaning which may be precisely defined in semantic terms of first-order logic. Semantics of first-order language provides an interpretation of the symbols used to construct sentences, so that these sentences become meaningful. It is rather exceptional that a clause itself entails its own meaning (P ← P, a tautologically true clause, and , the empty clause are such exceptions), because the logical value of the clause may depend on how one interprets the symbols which occur within it.

Formally, an interpretation for a set of clauses in a first-order language consists of a non-empty set called the domain (or world of discourse), an assignment of an element of the domain to each constant symbol, an assignment of a function on the domain to each function symbol, and an assignment of a relation on the domain to each predicate symbol. The role of the domain is to specify the set of individual objects which may have been named by terms of the language, or in other words, to specify the set of all possible values these terms may assume. Because in our case the language does not contain the equality symbol =, we restrict our considerations, without loss of generality, to interpretations which assign different values to different ground (i.e., variable-free) terms. This technical facility is known in the literature under the name of unique name assumption. It follows that an interpretation constitutes an idealized, but not necessarily finite, relational database.

Example 1. Consider a first-order language with the relation symbol Wife, and constant symbols Marek, Ewa, Raj, Radhika, and Gopi. We can interpret the relation symbol Wife in the usual way, i.e., Wife(a,b) means “a is the wife of b”. Consider the domain {Marek Suchenek, Ewa Suchenek, Rajshekhar Sunderraman, Radhika Venkataraman, Rajaraman Sunderraman}, which consists of five individuals. If we associate the constants Marek, Ewa, Raj, Radhika, and Gopi with the individuals Marek Suchenek, Ewa Suchenek, Rajshekhar Sunderraman, Radhika Venkataraman, and Rajaraman Sunderraman respectively, and associate the predicate symbol Wife with the relation {<Ewa Suchenek, Marek Suchenek>, <Radhika Venkataraman, Rajshekhar Sunderraman>}, then the following sentences are true in this interpretation:

1. Wife(Ewa,Marek)
2. (∃ x) Wife(Radhika,x)

However the sentence (∃ x) Wife(x,Gopi) is false under this interpretation.

An interpretation ι for a set of sentences S is said to be a model for S if each sentence of S is true in ι. In general, a set of sentences may have many essentially different models, for example, sizes of their relations may differ. If one treats interpretations as abstract relational databases, different sizes of relations correspond to different information content (the more tuples the greater the information content). It turns out that interpretations with possibly minimal information content, the so called minimal models, are the ones which are most useful in the context of database applications. More precisely, ι is called a minimal model if any deletion of tuples from the relations of ι infallibly results in an interpretation ι' which is not a model of S. A minimal model is the one which has possibly minimal relations, while satisfying all the sentences of S.

Example 2. The set of sentences:

S = {Male(x) V Female(x), Male(Marek), Female(Ewa), Cat(Fay)}

has two minimal models: one which interprets the predicate symbol Male as a unary relation {<Marek>}, and the predicate symbol Female as a unary relation {<Ewa>,<Fay>}, and another which interprets the predicate symbol Male as a unary relation {<Marek>,<Fay>}, and symbol Female as a unary relation {<Ewa>}. Models...
having larger relations, as for example one which interprets symbol Male as \{<Marek>,<Fay>\}, and symbol Female as \{<Ewa>,<Fay>\}, are not minimal. Interpretation of symbols Male and Female as \{<Marek>\} and \{<Ewa>\}, respectively, is not a model of \(S\) (does not satisfy the clause Male(x) V Female(x) which forces every x to be either male or female).

Semantics restricted to minimal models, the so called \textit{minimal model semantics}, is one of the central points of interest in deductive databases. It provides an adequate meaning for negative clauses. Its applications, however, go much beyond deductive databases: its variants have been widely accepted as the semantics in logic programming and in certain non-monotonic logics as well (under the name of predicate circumscription in the latter).

It is somewhat surprising that, although known from the early thirties, models for consistent sets of clauses may be built out of the ground terms of the language in which these clauses are expressed. This fact, discovered by the German mathematician Herbrand, remained as some kind of curiosity until its application in the seventies in the area of logics of programs and logic programming. Every Herbrand model has a unique domain which consists of all ground (i.e., variable-free) terms of the language in question, and assigns a ground term to itself as its value. For example, if 7 is a constant symbol and f is a function symbol, then the value of the term f(7) is f(7) in every Herbrand model. It is not true, however, that the value of f(x) is f(x) since x is a variable and therefore f(x) is not a \textit{ground} term. A value of a non-ground term in a Herbrand model, as in any other model, depends on the values of its variables. For instance, if the value of x is f(7) then the value of f(x) is f(f(7)).

Herbrand models are particularly useful in deductive databases because they allow semantic considerations of such databases to be restricted to purely syntactic objects. In the rest of the paper (and in Example 2), “minimal model” always means “minimal Herbrand model”.

**Deductive Databases**

As we have pointed out in the Introduction, deductive databases may be seen as generalizations of relational databases where, in addition to facts, general rules are allowed to be a part of the database. Let us consider the familiar “Suppliers and Parts” database in which the relation \(S\) represents information about suppliers, \(P\) represents information about parts, and \(SP\) represents information about which supplier supplies which part. Suppose, moreover, that this database is equipped with a view, product-by-location. A possible instance of this database is visualized in Figure 2.

**Figure 2: Suppliers-Parts Database**

This relational database can be turned into a deductive database by translating it into the following ground positive literals:

- \(S(S1,NYC)\)
- \(S(S2,NYC)\)
- \(S(S3,DC)\)
- \(P(P1,Nut)\)
- \(P(P2,Bolt)\)
- \(P(P3,Rivet)\)
- \(SP(S1,P1)\)
- \(SP(S1,P2)\)
- \(SP(S2,P2)\)
- \(SP(S3,P1)\)

and introducing the following clause:

\[\text{product-by-location}(x,y) \leftarrow S(y) \land SP(x,y).\]

One can query the deductive database based on the relations \(S\), \(P\), \(SP\) (base relations) and product-by-location (virtual relation).

Formally, a deductive database is a system of the form \(<\Sigma,\leftarrow>\), where \(\Sigma\) is a finite set of clauses and \(\leftarrow\) is a finite set of meta-rules for negative (or, in more general case, non-positive) inferences. \(\Sigma\) is comprised of ground literals (extensional part of a database) and other clauses (intensional part). If \(\Sigma\) contains exclusively definite clauses (or at least is logically equivalent to a set of definite clauses), then \(<\Sigma,\leftarrow>\) is called \textit{definite}. Otherwise, it is called \textit{indefinite}. \(\leftarrow\) describes how negative conclusion may be derived from \(\Sigma\). Having in \(\leftarrow\) suitable rules for negative inferences, one does not have to represent explicitly the negative information in \(\Sigma\), although integrity constraints may take a form of purely negative clauses (cf. (Jackson, 1988)).

Now, we turn to the notion of a query and its answer. Consider the query, “Find the supplier number and cities for suppliers who supply part \(P1\)”, over the suppliers-parts database mentioned earlier. It can be written as: \(\leftarrow S(x,y) \land SP(x,P1)\) and interpreted as: find all pairs of values \(x\) and \(y\) such that \(S(x,y)\) and \(SP(x,P1)\) can be derived from the database. The answer to this query is expressed in set notation as:
As another example, the query, “Find the parts that are supplied by suppliers located in NYC”, can be written as:

\{ \{ x,y \mid S(x,y) \land SP(x,P1) \} \}.

The answer to the first query can be easily verified to be \(<S1,NYC>\) and \(<S3,DC>\), and the answer to the second query can be easily verified to be \(<P1>\) and \(<P2>\). In obtaining the answer to these queries, we have made an implicit assumption that if a fact is not present in the database, we assume it to be false. We will make this concept more clear in the next section.

To see how one can obtain indefinite answers to queries, let us introduce the indefinite information \(SP(S4,P1) \lor SP(S5,P1)\) to the database along with the definite information \(S(S4,NYC)\) and \(S(S5,DC)\). Now, the answer to the first query, which asked for supplier numbers and cities for suppliers of part P1, includes the indefinite answer \(<S4,NYC> + <S5,DC>\), which expresses the fact that either S4 living in NYC, or S5 living in DC, supplies part P1. Recently, Liu and Sunderraman (1990) have generalized the relational model to represent indefinite information and provide an extended relational algebra to compute answers to queries.

### Closed World Assumptions

It was perhaps a need for efficient representation of information in a database that resulted in the emergence of the closed world assumption. If one needs to minimize the expected size of the representation, then there is a quite natural choice of the representation language: entries with high information content, as being the most likely (or desired) candidates for storing in a database, should have possibly the simplest syntactic form. If the predicate languages, customarily used for symbolic representation of information, this simplest form is materialized by atomic (and hence positive) statements. This natural convention, if successfully applied, has certain important consequences regarding the form of objectively true sentences expressible in the selected language. Since the amount of information in a statement is inversely proportional to its probability, or frequency of occurrence, it is the negative form of information that prevails. This is an idiosyncrasy of the very particular choice of the representation language, and would not necessarily hold if another selection criterion was utilized. It results in a hypothetical database containing a complete description of the real world in question consisting of a few positive facts in a deluge of negative information. Once the database designer realizes this fact, the solution to the problem of efficient representation becomes clear: only the positive facts (or, at least, non-negative) are physically represented in the database, while all the negative ones are implicitly assumed by default. This makes the essence of the closed world assumption in the sense of Reiter.

Formally, the closed world assumption may be defined as follows:

If an atomic sentence \(\Phi\) is not provable from the information contained in a database then assert \(\neg \Phi\).

**Example 3.** Suppose that the world of discourse is a collection of suppliers of raw materials and pending orders of such materials. A sample language suitable for representing the state of affairs in our world of discourse contains one binary predicate symbol \(Supplies\) and one 5-ary predicate symbol \(Order\), 7 constant symbols \(S_1, S_2, S_3, P_1, P_2, P_3, P_4\), and a collection of numeric constant symbols. The expression \(Supplies(s,p)\) means “Supplier s supplies part p” and the expression \(Order(n,p,s,m,a/bb/cc)\) means “There is an order with order number n for part p with supplier s for quantity q with a delivery date d”. Let us consider the following content of a database:

\[
\begin{align*}
Supplies(S_1, P_1) & \quad Order(1, P_1, S_2, 100, 1/30/90) \\
Supplies(S_2, P_2) & \quad Order(2, P_2, S_2, 150, 6/30/90) \\
Supplies(S_2, P_3) & \quad Order(3, P_3, S_2, 750, 3/15/90) \\
Supplies(S_3, P_1) & \quad Order(4, P_3, S_2, 750, 9/15/90) \\
Supplies(S_4, P_1) & \quad Order(5, P_4, S_5, 500, 12/15/90).
\end{align*}
\]

The statement “There is a supplier who supplies both \(P_1\) and \(P_2\)” (namely \(S_1\)) is a logical consequence of the database. The statement “\(P_1\) has not been ordered” is not a logical consequence of the database, however, it follows from the database under the closed world assumption. This coincides with the natural interpretation of the content of the database. To store all the negative information implied by the closed world assumption one has to add all the negative statements of the form, \(\neg Supplies(S_i, P_j)\), and a prohibitively large number of statements of the form, \(\neg Order(n, P_i, S_j, m, a/bb/cc)\). Besides the waste of database memory, the usefulness of statements like \(\neg Order(7348, R_3, S_1, 57942, 1/29/90)\) is at best doubtful, since its information content is next to zero.

If our database is supposed to admit information about the current status and timely realization of orders, then a new unary predicate symbol \(Delay\) may be necessary, where \(Delay(x)\) means “Order with order number x is delayed”. If we assume that \(Delay(2)\) is an entry in the database then under the closed world assumption we can infer \(\neg Delay(1), \neg Delay(3), \text{ etc.}\), without the necessity of representing these facts in the database. In the case where most of the orders are subject to a delay the statement \(\neg Delay(x)\)
Delay(1), if true, is more informative than Delay(1), and therefore a different extension of the database is more appropriate: instead of Delay, another unary predicate symbol On-Time may be used. Although On-Time(1) has, formally, the same meaning as ~ Delay(1), in the case where the order with order number 1 is the only order not subject to a delay (which means that the probability of delay is, say, around 80%), under the closed world assumption one statement On-Time(1) substitutes for 4 statements Delay(2), ..., Delay(5).

The above example clearly shows that for the purpose of effective minimization of the size of symbolic representation, the choice of representation language has to depend on the probability distribution in the space of representable events.

Reiter’s closed world assumption is, in the case of a reasonable choice of representation language, equivalent to asserting by default facts which are most likely to happen. This natural scheme, however, is known to lead to a contradiction when applied to indefinite databases. For example, applying the closed world assumption to a database consisting of one entry: Supplies(S1,P1) ∨ Supplies(S3,P1)

one can infer both ~ Supplies(S1,P1) (since Supplies(S1,P1) is not provable from the database) and ~ Supplies(S3,P1) (same reason as before), or by de Morgan’s law: (~ Supplies(S2,P1) ∨ ~ Supplies(S3,P1)), a conclusion that clearly contradicts the actual content of the database. Therefore, the proper treatment of default negative information requires a more subtle technique for indefinite databases.

One of the best proven methods of avoiding inconsistency in a formal reasoning system is to provide it with a precise meaning, so that the formal deductions allowed in this system preserve all possible meanings for the premises used in such deductions. As we have seen before, first-order logic has a well defined semantics, which — by the virtue of the completeness theorem — has such preservation property: a sentence P is provable from a set of premises S if and only if P is true in every model of S. Finding an appropriate semantics for the closed world assumption is our first step towards its consistent generalization over indefinite databases.

Let us take a closer look at the semantic effects of applying the closed world assumption to a definite database DB of the form {P(C1, ..., Ck), P(D1, ..., Dl), ...}. What the closed world assumption says is that the only tuples which belong to the relation P are those explicitly specified in DB, i.e., <C1, ..., Ck, D1, ..., Dl> etc. Of course, DB itself proves that these tuples are in P, but it does not prove, in the case of absence of negative information, that no other tuples (namely, those not specified in DB, e.g., <A1, ..., An>, are not in P. Thus the closed world assumption applied to DB describes (unambiguously) P as the least relation containing all the tuples listed in DB.

From this observation we conclude that the closed world assumption restricts the meaning of the database in question to a collection of relations that are minimal (in the sense of set-theoretic containment). Therefore, minimal models are the ones which provide the semantics of the closed world assumption, and allow for its generalization over indefinite databases. The following definition of minimal entailment constitutes the intentional semantics of this generalization.

A clause Φ is minimally entailed by a database if and only if Φ is true in all minimal models of DB.

Example 4. The definite database: Supplies(S1,P1)
Supplies(S2,P2)
Supplies(S3,P3)

Supplies(S1,P1) has 27 models (i.e., 27 different Sis and Pjs, i,j = 1,2,3, satisfy all statements of this database). For example, the set ALL = {<S,P> | i,j = 1,2,3} of all combinations of Sis and Pjs is a model of this database.

Exactly one of them, namely:

{<S1,P1>,<S2,P1>,<S2,P2>,<S3,P3>}

is the minimal model of the database. The clause ~ Supplies(S1,P1) follows from this database under the closed world assumption. This clause is true in the minimal model of this database, i.e., it is minimally entailed by this database. However, this clause is not true in certain (non-minimal) models of this database, e.g., it is not true in the model ALL mentioned above. Therefore ~ Supplies(S1,P1) is not provable from this database.

Example 5. The indefinite database:

Supplies(S1,P1)
Supplies(S2,P2)
Supplies(S3,P3)

Supplies(S2,P1) V Supplies(S3,P3)

has 3 x 27 models. For example, the set ALL of the previous example is a model of this database. Two of them, namely {<S1,P1>,<S2,P1>,<S3,P3>} and {<S3,P1>,<S3,P2>,<S3,P3>} are minimal models of this database. The clauses ~ Supplies(S1,P1) and ~ Supplies(S2,P1) follow from this database under the closed world assumption. Neither of them, however, is minimally entailed by this database: ~ Supplies(S1,P1) is false in the first minimal model (because Supplies(S1,P1) is true in this model), and ~ Supplies(S2,P1) is false in the second minimal model of this database.

The minimal entailment makes the semantic equivalent of the closed world assumption for indefinite databases. It does not seem, however, at least as it is, a particularly useful substitute for the closed world assum-
tion. One can expect that the combinatorial explosion of models practically precludes an efficient method of enumeration of all the minimal models of an indefinite database, making the problem: “Is $\Phi$ true in all minimal models of the database?” rather hard for direct verification. Finding an appropriate syntactic characterization of this concept by means of a derivation procedure may bring us closer to a computationally feasible solution of this problem, or at least provide a proof-like technique for verification of automated reasoning in the context of minimal model semantics.

The question of a pure syntactic characterization of minimal entailment remained open for almost a decade. Although a generalized closed world assumption GCWA proposed by Minker, followed by Yahya-Henschen’s GCWAC (GCWA for Conjunctions) (Yahya & Henschen, 1985), provided a partial answer to this question, and has been known to avoid the inconsistency paradox of Reiter’s version of the closed world assumption, it did not completely solve the problem of characterization for disjunctive or quantified queries. Finally, this problem received a surprisingly simple answer in terms of a certain modification $cwa_5$ of the closed world assumption, introduced by Suchenek (1987). Since, as has been observed before, limitation of the explicit content of a database to positive information has been recognized as an efficient means of minimization of the physical size of the database, one can suspect that any positive statement not provable from the database was not intended as its implicit consequence. Of course, it was a matter of proof (Suchenek, 1989), nevertheless it seems intuitively acceptable that the following version $cwa_5$ of the closed world assumption provides a complete syntactic characterization of the minimal entailment:

If adding clause $\Phi$ to a database DB does not enlarge the set of positive logical consequences

One may show by not so difficult inspection that for quantifier-free clause $\Phi$ and definite database DB, $cwa_5 + DB$ proves $\Phi$ if and only if $cwa + DB$ proves $\Phi$. For more complex clauses and for indefinite databases, this equivalence may not be true. In particular, $cwa_5$, as having semantic counterpart (namely, the minimal entailment), may never lead to a contradiction. This is an important feature that Reiter’s $cwa$ does not possess. It also has been shown in (Suchenek, 1989) that Minker’s GCWA may be equivalently expressed by restricting the scope of the clause $\Phi$ in the definition of $cwa_5$ to atomic and negated atomic sentences.

Example 6. Let us recall the example of a database DB: $\text{Supplies}(S_2,P_1) \lor \text{Supplies}(S_3,P_1)$. We demonstrated that Reiter’s $cwa$ is inconsistent, namely the database under the $cwa$ proves both $\neg \text{Supplies}(S_2,P_1)$ and $\neg \text{Supplies}(S_3,P_1)$. Let us see why $cwa_5$ does not lead to a contradiction when applied to this database. Is $\neg \text{Supplies}(S_2,P_1)$ implied by DB under $cwa_5$? To answer this question, we have to consider positive logical consequences of $DB' = \neg \text{Supplies}(S_2,P_1) + DB$. We have that $DB'$ logically implies: $\neg \text{Supplies}(S_2,P_1) \land (\text{Supplies}(S_2,P_1) \lor \text{Supplies}(S_3,P_1))$, that is to say, $DB'$ implies $\text{Supplies}(S_3,P_1)$. This is a positive clause which is not provable from DB itself. Therefore, by the definition of $cwa_5$, $DB + cwa_5$ does not assert $\neg \text{Supplies}(S_3,P_1)$. Similarly, one can verify that $cwa_5 + DB$ does not assert $\neg \text{Supplies}(S_3,P_1)$ either. The inconsistency caused by Reiter’s $cwa$ has disappeared.

The closed world assumption in any of its forms discussed above ($cwa$, GCWA, $cwa_5$) minimizes evenly all the relation symbols of the database’s language, which is not particularly desirable if the database in question consists of conceptual layers built one upon another. If, for example, a deductive database is a result of translation of a certain relational database with incorporated views (as for instance, the database visualized in Figure 2), then the image of the physical level and images of these views may not necessarily need a uniform treatment. To the contrary, one can expect that relations which belong to the images of more primitive views should be minimized before the others. This observation brings us to the problem of prioritized minimization.

There are several possible approaches to incorporate priorities into minimal model semantics. In some logic programs, as we will see in the next Section, such priorities are implicitly introduced by certain preference relation, defined in class of minimal models, induced by a form of logic program in question. In more general artificial intelligence context, prioritized circumscription tackles this problem from a position of power, or in other words, second order logic.

Here, we demonstrate how $cwa_5$ may be successfully used to allow for prioritized minimization. We start from a modified Clark’s example, the original version of which was analyzed in (Clark, 1978).

Example 7. Let D be a deductive database with the extension:

$D' = \{ \text{Student}(J.Brown) \leftarrow \text{Student}(D.Smith) \leftarrow \text{Takes}(J.Brown,C101) \leftarrow \text{Takes}(D.Smith,C101) \leftarrow \text{Takes}(D.Smith,C301) \leftarrow \}$
Math-Course(C101) ←
Math-Course(C301) ←
(which may be interpreted as an image of the physical level of a relational database) and the intention:
\[ D_i = \{ \text{Non-Math-Major}(x) \lor \text{Takes}(x,y) \leftarrow \text{Student}(x) \land \text{Math-Course}(y) \} \]
(which may be interpreted as an image of a view of this relational database), i.e., let \( D = D_e \cup D_i \).

\( D_i \) lists students, courses and enrollments. \( D_i \) says that a student who does not take all listed math courses is a non-math major. This database has two classes of minimal models: one, in which Non-Math-Major(J.Brown) is true and Takes(J.Brown,C301) is false, and the other, in which Non-Math-Major(J.Brown) is false and Takes(J.Brown,C301) is true. It seems clear that only the first class of models is adequate to intuitive comprehension of the content of the database \( D \). In particular, one can expect that \( D \) implies Non-Math-Major(J.Brown). However, \( D \) does not minimally entail Non-Math-Major(J.Brown) since this sentence is false in the second class of minimal models of \( D \). Therefore, cwa\(_{s}\) + \( D \) does not prove Non-Math-Major(J.Brown).

If a deductive database in question is indeed a result of translating a physical level of relational database with several views implemented on it, then the proper cure of the above pathological situation is quite simple: first, the image of the physical level should be minimized, and only after that images of views should undergo subsequent minimization. To achieve this effect using cwa\(_{s}\), one has to apply it several times to the appropriately layered database, starting from the core, corresponding to the physical level, and proceeding from inner to outer layers. In the above example, this proper mode of prioritized minimization is to first minimize simultaneously the predicates Student, Takes and Math-Course, and after that to minimize the predicate Non-Math-Major. This kind of minimization is achieved by a double application of cwa\(_{s}\): cwa\(_{s}\)(\( D_i \cup \text{cwa}_{s}(D_e) \)).

Let us verify that this proves Non-Math-Major(J.Brown). First, let us show that cwa\(_{s}\)(\( D_e \)) proves ~Takes(J.Brown,C301). Indeed, all positive ground clauses provable from \( D_e \cup \{ \sim \text{Takes}(J.Brown,C301) \} \) (within the language of \( D_e \)) are already provable from \( D_i \). Since in every Herbrand model of \( D \), universal quantification \( \forall x \ \Phi(x) \) is equivalent to: \( \Phi(J.Brown) \land \Phi(D.Smith) \land \Phi(C101) \land \Phi(C301) \), from this we infer that all positive clauses (not only ground ones) provable from:
\[ D_e \cup \{ \sim \text{Takes}(J.Brown,C301) \} \]
are already provable from \( D_i \). Now, \( D_i \cup \text{cwa}_{s}(D_e) \) proves Non-Math-Major(J.Brown), because \( D_i = D_e \cup \{ \sim \text{Takes}(J.Brown,C301) \} \) does. Therefore, cwa\(_{s}\)(\( D_i \cup \text{cwa}_{s}(D_e) \)) proves the same statement too. It may be noted that the second application of cwa\(_{s}\), which ensures minimization of predicate Non-Math-Major, was actually not necessary for proof of Non-Math-Major(J.Brown). It is necessary, however, for proof of \( \sim \)Non-Math-Major(D.Smith).

In a more complex scenario, appropriate precaution should be taken while minimizing layers of deductive databases. For example, if \( D \) is an image of a relational database of the form in Figure 3, say, \( D_0 \) is an image of the physical level, and \( D_1, D_2, D_3 \) are images of view\(_1\), view\(_2\), view\(_3\), respectively, then appropriate prioritized minimization of \( D \) is achieved by:
\[ \text{cwa}_{s}(D_3 \cup (\text{cwa}_{s}(D_1 \cup \text{cwa}_{s}(D_2)))) = \text{cwa}_{s}(D_3 \cup \text{cwa}_{s}(D_2)) \]\
Constructions of the above form, although intuitively obvious, are currently under investigation (e.g., in forthcoming (Suchenek & Sunderraman, 1990)) and require further research.

**Deductive Databases versus Logic Programming**

The areas of deductive databases and logic programming are so similar that the relationship between them cannot go unmentioned. These two areas started differently and there used to be separate research activities pertinent to each of them. On the other hand, there has been a substantial interplay between the two areas, as the inspirations, ideas, and results from one area infiltrated the other. This migration of methodology caused certain confusion among the general audience, which has been
amplified by a lack of clear widely accepted taxonomy. In particular, the rather extensive use of the term “logic” in the context of deductive databases, as in “logic databases”, contributed to the not so uncommon misuse of terminology. So, it is actually the subtle difference between the two areas which requires an explanation. To see the difference from a proper perspective, however, we must start by examining the similarities.

Similar to deductive databases, logic programs are composed of a finite number of clauses (actually deductive databases borrowed this form from logic programming). Originally, they were required to consist exclusively of definite clauses (van Emde & Kowalski, 1976), however, recently the general logic programs with indefinite clauses have also been considered useful (Minker & Rajasekar, 1989). Therefore, as far as the first-order expressive power is concerned, there is no essential difference between logic programs and deductive databases, although a typical logic program will, probably, contain considerably fewer ground clauses (facts) than a typical deductive database. Moreover, indefinite clauses allowed in deductive databases have a different form, namely:

\[ A_1 \vee \ldots \vee A_n \leftarrow B_1, \ldots, B_m \]

than indefinite clauses allowed in logic programs:

\[ A_1 \leftarrow B_1, \ldots, B_m, \ldots, A_n, \]  

\[ \ldots, A_n, \]

where instead of disjunctive heads, negation in the body may occur.

Similar to deductive databases, in logic programs negation is weaker than in first-order logic: for every positive literal \( P \), if \( \neg P \) is provable from a consistent program in the classical sense then \( \neg P \) will also be derived using any of the existing sound negation as finite failure procedures, but not necessarily vice versa. Moreover, in the case of definite databases and logic programs, the intended meaning of negation is, as we will see, the same for both.

The last similarity brings us to one of the essential differences between logic programs and deductive databases: in the general indefinite case, their semantics do not coincide. Semantics for logic programs is operational, defined in terms of resolution refutation procedure with negation treated as finite failure to prove, while semantics for deductive databases is denotational, based on the notion of minimal model. As a result, negation in indefinite deductive databases gets a different meaning than in indefinite logic programs. Moreover, this difference is by no means negligible; semantics of indefinite logic programs can distinguish between logically equivalent clauses, as opposed to deductive databases, and of course to first-order logic. This is a result of different rules for inferring negative conclusions in deductive databases and logic programming.

As opposed to deductive databases, where various versions of the closed world assumption are used, in logic programs the following rule due to Clark (1978) is applied to derive negative ground literal \( \neg L \):

If the execution (by means of resolution refutation procedure) of program \( P \) on goal \( L \) finitely fails (i.e., it is evident that \( L \) cannot be achieved, even if the Clark’s rule is subsequently used), then assert \( \neg L \).

This negation as finite failure rule is, to all appearances, very similar to Reiter’s version of cwa, where \( \neg L \) is asserted if \( L \) cannot be proved. It should be mentioned that “derivability” in case of negation as finite failure means, somewhat recursively, derivability using possibly negation as finite failure, while in case of cwa, it is pure first-order provability. Let us consider the following.

**Example 8.** The clause:

\[ \text{Father(Pat,Gary)} \leftarrow \text{Parent(Pat,Gary)} \land \neg \text{Mother(Pat,Gary)} \]

and the clause:

\[ \text{Mother(Pat,Gary)} \leftarrow \text{Parent(Pat,Gary)} \land \neg \text{Father(Pat,Gary)} \]

are indefinite program clauses, logically equivalent to the following disjunctive clause:

\[ \text{Father(Pat,Gary)} \lor \text{Mother(Pat,Gary)} \]

Parent(Pat,Gary) and therefore equivalent to each other. The programs:

\[ P = \{ \text{Father(Pat,Gary)} \land \neg \text{Mother(Pat,Gary)} \} \]

Parent(Pat,Gary) \leftarrow \}

and

\[ Q = \{ \text{Mother(Pat,Gary)} \land \neg \text{Father(Pat,Gary)} \} \]

Parent(Pat,Gary) \leftarrow \}

are perfectly legal indefinite logic programs. Although \( P \) and \( Q \) are logically equivalent (in the sense that one is provable from the other within first-order logic), they do not return the same answer to the query

\[ \leftarrow \text{Father(Pat,Gary)} \]

In particular, program \( P \) returns the answer “yes” and program \( Q \) returns the answer “no”. In case of program \( Q \), this query cannot be matched, or more precisely unified, with a head of any clause of \( Q \), therefore the attempts to prove \( \text{Father(Pat,Gary)} \) fails finitely (namely in one step). Negation as finite failure rule yields \( \neg \text{Father(Pat,Gary)} \). In case of program \( P \), this query trivially unifies with the head of the first clause of \( P \), which subsequently causes the sub-queries \( \text{Parent(Pat,Gary)} \) and \( \neg \text{Mother(Pat,Gary)} \) to be issued. Both of these sub-queries are answered “yes”, the first one after trivial unification and the second one after finite failure of proving \( \text{Mother(Pat,Gary)} \).

The overview nature of this paper does not allow us to discuss the technicalities necessary for full explanation of the above phenomenon. To give an idea, however, why this is so, consider the set of minimal models for \( P \) and

---

*Journal of Database Administration*
Q. Both programs are equivalent to the following sentence:
\[ ([\text{Father}(\text{Pat}, \text{Gary}) \lor \text{Mother}(\text{Pat}, \text{Gary}) \leftarrow \text{Parent}(\text{Pat}, \text{Gary})]) \land \neg \text{Parent}(\text{Pat}, \text{Gary}) \]
or, in other words, to:
\[ \text{Father}(\text{Pat}, \text{Gary}) \lor \text{Mother}(\text{Pat}, \text{Gary}). \]
Because equivalent sentences have the same models, in particular the same minimal models, the minimal models for \( P \) coincide with the minimal models for \( \text{Father}(\text{Pat}, \text{Gary}) \lor \text{Mother}(\text{Pat}, \text{Gary}) \), and of course coincide with the minimal models for \( Q \). The entire class of minimal models for \( \text{Father}(\text{Pat}, \text{Gary}) \lor \text{Mother}(\text{Pat}, \text{Gary}) \) may be split onto two disjoint classes: models in which \( \text{Father}(\text{Pat}, \text{Gary}) \land \neg \text{Mother}(\text{Pat}, \text{Gary}) \) is true, and models in which \( \neg \text{Father}(\text{Pat}, \text{Gary}) \land \text{Mother}(\text{Pat}, \text{Gary}) \) is true. It turns out that the first class makes the semantics of the program \( P \), while the second one makes the semantics of the program \( Q \). For the logically equivalent deductive database:
\[ D = \{ \text{Father}(\text{Pat}, \text{Gary}) \lor \text{Mother}(\text{Pat}, \text{Gary}) \leftarrow \text{Parent}(\text{Pat}, \text{Gary}) \land \neg \text{Parent}(\text{Pat}, \text{Gary}) \} \]
the entire union of the two classes constitutes a valid semantics. This means that a form of a logic program encodes certain relation of preference, which favors some minimal models (e.g., those satisfying \( \text{Father}(\text{Pat}, \text{Gary}) \land \neg \text{Mother}(\text{Pat}, \text{Gary}) \) over others (in this case, those satisfying \( \neg \text{Father}(\text{Pat}, \text{Gary}) \land \text{Mother}(\text{Pat}, \text{Gary}) \)).

Since every set of definite clauses, and therefore every definite logic program has exactly one minimal model, the least Herbrand model, the relation of preference becomes irrelevant in such a case (its domain has one element to choose from). Therefore, definite logic programs and definite deductive databases are equivalent from a semantic point of view. In particular, negation has exactly the same meaning in both.

It follows from Example 8 that different but logically equivalent logic programs may have different meanings, because they may induce different preference relations in the class of their minimal models. If one hopes that the set of all logic programs equivalent to a deductive database in question captures the real meaning of this database, then one is wrong. The following example shows that logic programs can hardly be accepted as operational implementations of indefinite deductive databases.

Example 9. Let us consider the deductive database:
\[ D = \{ \text{male}(x) \lor \text{female}(x) \leftarrow \text{human}(x), \text{human}(\text{Raj}) \}, \text{human}(\text{Radhika}) \} \]
There are two logic programs equivalent to \( D \):
\[ P = \{ \text{male}(x) \leftarrow \text{human}(x) \land \neg \text{female}(x), \text{human}(\text{Raj}) \} \]
\[ Q = \{ \text{female}(x) \leftarrow \text{human}(x) \land \neg \text{male}(x), \text{human}(\text{Raj}), \text{human}(\text{Radhika}) \} \]
Neither of these two programs allow the clause:
\[ \text{male}(\text{Raj}) \land \text{female}(\text{Radhika}) \]
to be true. What is worse is that every computation of \( P \) and \( Q \) proves:
\[ \text{male}(\text{Raj}) \land \text{male}(\text{Radhika}). \]
Also, the program \( R = P \cup Q \) will never stop on query containing male or female. Of course, in a certain minimal model of \( D \), both male(\text{Raj}) and female(\text{Radhika}) hold.

There are, of course, other differences between deductive databases and logic programs, but since negation constitutes the main subject of this paper, we briefly mention only the following two:
1. In deductive databases, queries with free variables, e.g., \( P(x,y) \) are answered by a set of tuples of ground terms, e.g., \( \{ \langle t_1, s_1 \rangle, \ldots, \langle t_n, s_n \rangle \} \), satisfying the query. In logic programs, goals are universally quantified (implicitly), therefore the only outcome of computation may be true or false. In this respect queries directed to deductive databases are strictly more expressive than goals directed to logic programs.
2. In deductive databases a separate class of issues and techniques are associated with updates, which do not belong to the typical domain of logic programs. Consequently, such activities, pertinent to databases, as integrity enforcement, do not belong to logic programming.

Figures 4 and 5 summarize the similarities and differences between deductive databases and logic programs.

**Perspectives**

Since the fields of deductive databases and logic programming are intimately related, it is difficult to categorize some of the important landmarks as belonging to one field or the other. Both fields seem to have their roots in the area of automated theorem proving.

In 1965, Robinson (1965) introduced the resolution principle, a uniform method for performing automated deduction. This was a significant breakthrough as the resolution principle was a highly efficient proof procedure and was easily implementable on computers. This principle is used by automated deduction systems, the so called theorem provers. Using the resolution principle, Green and Raphael (1968) designed the system QA3.5 which showed the feasibility of implementing deductive databases in a uniform manner. This work can be thought of as the starting point of the field of deductive databases.

In 1977, the workshop on Logic and Databases (Gallaire & Minker, 1978) produced a number of important papers that established deductive databases as a
legitimate field on its own. Clark’s paper on negation as failure and completed database introduced the meta-rule, negation as finite failure, to solve the problem of inferring negative information from a deductive database, in a logic programming context (Clark, 1978). In his paper, deductive databases were treated as a special case of logic programs with a large number of facts and only a few rules. This conjecture, however, has not been widely accepted, and as we have indicated there are essential differences between deductive databases and logic programs (cf. (Apt, 1988), section 8.3, deductive databases). Consequently, negation as failure and its denotational equivalent, program completion, have finally found their place in logic programming.

Reiter’s paper on the closed world assumption discussed the notion of inferring negative information from deductive databases with clear motivations from the relational model of a database (Reiter, 1978b). He also introduced the notion of a query and answers to queries in deductive databases. The closed world assumption may be understood as the database counterpart of Clark’s negation as failure rule. In 1982, Minker introduced the generalized closed world assumption which extends cwa over indefinite deductive databases (Minker, 1982). Re-

In the logic programming front, Kowalski (1974) proposed the use of logic as a programming language. The paper by van Emde and Kowalski (1976) introduced the notion of unique minimal model of a set of definite clauses and used a fixed-point operator to characterize the meaning of definite logic programs. Apt and van Emde (1982) characterized linear resolution with selection function in definite logic programs with negation as finite failure (SLDNF) in terms of the fixed-point operator. Jaffer, Lassez, and Lloyd (1983) proved the completeness of negation as finite failure. An excellent introduction to logic programming is found in (Apt, 1988). Lloyd (1987) is another source of the developments in the field of logic programming.

Although most of the work in deductive databases and logic programming has been done over the last two decades, the notion of closed world assumption has been used implicitly in automated reasoning systems well before deductive databases. In particular, FORTRAN compilers of late 1950s actually used the concept when solving a system of equivalences defined by EQUIVALENCE statements.

At present there are no commercial deductive database systems. However, there are a number of experimental projects that may result in commercial systems in the near future. Below, we mention three such projects. NAIL! (Not Another Implementation of Logic!) is an experimental deductive database system under development at Stanford University (Morris, Ullman, & Van Gelder, 1986; Ullman, 1985). LDL (Logic Data Language) is another deductive database system and language under development at Microelectronics Computer Consortium (MCC), in Austin, Texas (Chimenti et al., 1987; Naqvi and Tsur, 1988; Zaniolo, 1988). POSTGRES is a successor to the INGRES project under development at the University of California at Berkeley (Stonebraker and Rowe, 1986a, 1986b; Wensel, 1988). Ullman (1988) contains detailed discussion of the various techniques used in most of these systems.

Acknowledgements

It was Jack Minker’s Distinguished Lecture given at Wichita State in Fall 1989 that inspired us to write this paper. We also would like to express sincere appreciation to our colleagues, Mary Edgington and John Hutchinson, for their valuable suggestions and corrections of our ESL. Xitong Zheng helped us in our struggles with LaTeX and his assistance is greatly appreciated. Separate thanks are due to Mohammad Dadashzadeh for his stimulating enthusiasm and helpful comments.

REFERENCES


Journal of Database Administration

NY: Springer-Verlag.


Marek A. Suchenek is an associate professor and chairperson of the Computer Science Department at The Wichita State University. He has authored or co-authored two books and over 30 papers covering a broad spectrum of computing sciences, including logical foundations of artificial intelligence, program correctness and verification, and testing real-time systems.

Rajshekhar Sunderraman is an assistant professor of computer science at The Wichita State University. He has authored or co-authored several papers in the areas of incomplete information in databases and deductive databases. His research interests include deductive databases, incomplete information, and logic programming.
Related Content

GeoBase: Indexing NetCDF Files for Large-Scale Data Analysis
[www.irma-international.org/chapter/geobase/85460/](www.irma-international.org/chapter/geobase/85460/)

The Expert's Opinion
[www.irma-international.org/article/expert-opinion/51112/](www.irma-international.org/article/expert-opinion/51112/)

INDUSTRY AND PRACTICE: A Metadata Management System to Support Data Interoperability, Reuse and Sharing
[www.irma-international.org/article/industry-practice-metadata-management-system/51134/](www.irma-international.org/article/industry-practice-metadata-management-system/51134/)

[www.irma-international.org/article/industry-practice-understanding-three-dimensions/51157/](www.irma-international.org/article/industry-practice-understanding-three-dimensions/51157/)

The MAIS Approach to Web Service Design
[www.irma-international.org/chapter/mais-approach-web-service-design/4393/](www.irma-international.org/chapter/mais-approach-web-service-design/4393/)