Deductive Database Languages: Problems and Solutions

MENGCHI LIU

University of Regina

Deductive databases result from the integration of relational database and logic programming techniques. However, significant problems remain inherent in this simple synthesis from the language point of view. In this paper, we discuss these problems from four different aspects: complex values, object orientation, higher-orderness, and updates. In each case, we examine four typical languages that address the corresponding issues.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages—*Data description languages* (DD); *Data manipulation languages* (DM); *Query languages*; *Database (persistent) programming languages*; H.2.1 [**Database Management**]: Logical Design—*Data models*; *Schema and subschema*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*Deduction* (e.g., natural, rule-base); *Logic programming*; *Nonmonotonic reasoning and belief revision*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Representation languages*; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.3.2 [**Programming Languages**]: Language Classifications; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Logic and constraint programming*

General Terms: Languages, Theory

Additional Key Words and Phrases: Deductive databases, logic programming, nested relational databases, complex object databases, inheritance, object-oriented databases

1. INTRODUCTION

Databases and logic programming are two independently developed areas in computer science. Database technology has evolved in order to effectively and efficiently organize, manage and maintain large volumes of increasingly complex data reliably in various memory devices. The underlying structure of databases has been the primary focus of research that has led to the development of various data models. The most well-known and widely used data model is the relational data model [Codd 1970]. The power of the relational data model lies in its rigorous mathematical foundations with a simple user-level paradigm and set-oriented, high-level query languages. However, the relational data model has been found inexpressive for many novel database applications. During the past decade, many

Author's address: Department of Computer Science, University of Regina, Regina, Saskatchewan S4S 0A2, Canada; email: mliu@cs.uregina.ca.

CONTENTS

more expressive data models and systems have been developed, such as the ER model [Chen 1976], RM/T [Codd 1979], TAXIS [Mylopoulos et al. 1980], SDM [Hammer and McLeod 1981], DAPLEX [Shipman 1981], SHM+ [Brodie 1984], Gemstone [Maier et al. 1986], Galileo [Albano et al. 1985], SAM*[Su 1986], Iris [Fishman et al. 1987], IFO [Abiteboul and Hull 1987], the one for EXODUS [Carey et al. 1988], Orion [Kim 1990a; 1990b], O_2 [Deux et al. 1991], Jasmine [Ishikawa et al. 1993], Fabonacci [Albano et al. 1995], ODMG [Cattell 1996], and others.

Logic programming is a direct outgrowth of earlier work in automatic theorem proving and artificial intelligence. It is based on mathematical logic, which is the study of the relations between assertions and deductions and is formalized in terms of proof and model theories. Proof theory provides formal specifications for correct reasoning with premises, while model theory prescribes how general assertions may be interpreted with respect to a collection of specific facts. Logic programming is programming by description. It uses logic to represent knowledge and uses deduction to solve problems by deriving logical consequences. The most well-known and widely used logic programming language is Prolog [Colmerauer 1985; Kowalski 1988; Sterling and Shapiro 1986], which uses the Horn clause subset of first-order logic as a programming language and the resolution principle as a method of inference with well-defined model-theoretic and proof-theoretic semantics [Lloyd 1987]. However, Prolog lacks expressiveness in a number of areas. Over the past several years, a number of richer and more expressive logic programming languages have been proposed and implemented, such as LOGIN [Aït-Kaci and Nasr 1986], LIFE [Aït-Kaci and Nash 1986], HiLog [Chen et al. 1993], Prolog++ [Moss 1994], Gödel [Hill and Lloyd 1994], Oz [Smolka 1995], Mercury [Somogyi et al. 1996], XSB [Rao et al. 1997], etc.

Important studies on the relations between logic programming and relational databases have been conducted for about two decades, mostly from a theoretical point of view [Gallaire and Minker 1978; Gallaire 1981; Jacobs 1982; Ullman 1982; Maier 1983]. Relational databases and logic programming have been found quite similar in their representation of data at the language level. They have also been found complementary in many aspects. Relational database systems are superior to the standard implementations of Prolog with respect to data independence, secondary storage access, concurrency, recovery, security and integrity [Tsur and Zaniolo 1986]. The control over the execution of query languages is the responsibility of the system, which uses query optimization and compilation techniques to ensure efficient performance over a wide range of storage structures. However, the expressive power and functionality of relational database query languages are limited compared to that of logic programming languages. Relational languages do not have built-in reasoning capabilities. Besides, relational query languages are often powerless to express complete applications, and are thus embedded in traditional programming languages, resulting in *impedance mismatch* [Maier 1987] between programming and relational query languages. Prolog, on the other hand, can be used as a general-purpose programming language. It can be used to express facts, deductive information, recursion, queries, updates, and integrity constraints in a uniform way [Reiter 1984; Sterling and Shapiro 1986].

The integration of logic programming and relational database techniques has led to the active research area of deductive databases [Gallaire et al. 1984; Ceri et al. 1990; Grant and Minker 1992]. This combines the benefits of the two approaches, such as representational and operational uniformity, reasoning capabilities, recursion, declarative querying, efficient secondary storage access, etc. The function symbols of Prolog, which are typically used for building recursive functions and complex data structures, have not been found useful for operating over relational databases made up of flat relations. As a result, a restricted form of Prolog without function symbols called Datalog (with negation), with a well-defined declarative semantics based on the work in logic programming, has been widely accepted as the standard deductive database language [Ullman 1989a; Ceri et al. 1990].

In the past few years, various set-oriented evaluation strategies specific for deductive databases have been the main focus of extensive research [Bancilhon et al. 1986; Bancilhon and Ramakrishnan 1986; Beeri and Ramakrishnan 1991; Ceri et al. 1990; Ioannidis and Ramakrishnan 1988; Jiang 1990; Mumick et al. 1996; Sacca and Zaniolo 1987; Ullman 1989a; Ullman 1989b] and a number of deductive database systems or prototypes based on Datalog have been reported. These include Nail [Morris et al. 1986], LOLA [Freitag et al. 1991], Glue-Nail [Derr et al. 1993], XSB [Sagonas et al. 1994], CORAL [Ramakrishnan et al. 1994], Aditi [Vaghani et al. 1994], LogicBase [Han et al. 1994], Declare/SDS [Kiebling et al. 1994] etc. See Ramakrishman and Ullman [1995] for a survey of these deductive database systems. Deductive database systems have been used in a variety of application domains including scientific modeling, financial analysis, decision support, language analysis, parsing, and various applications of transitive closure such as bill-of-materials and path problems [Ramakrishman 1994]. They are best suited for applications in which a large amount of data must be accessed and complex queries must be supported.

However, significant problems remain inherent in this kind of deductive database from the language point of view. In this paper, we investigate four problematic areas. The first one is that such deductive databases provide inexpressive flat structures and cannot directly support the complex values common in novel database applications. The second is that object orientation offers some of the most promising ways of meeting the demands of the most advanced database applications, but it is not clear how to incorporate object orientation into deductive databases to increase their data modeling capability. The third is that current deductive databases cannot naturally deal with schema and higher-order features in a uniform framework so that a separate, non-logic-based language is normally provided to specify and manipulate the schema information. Finally, it is necessary to be able to update and modify a database, but there is no immediately obvious way of expressing updates within a logical framework. We elaborate these problems and survey four typical languages for each area with consistent examples.

There are other significant issues relevant to deductive database languages. These include semantics of logic programs with negation such as predicate stratification and fixpoint semantics [Apt et al. 1988], local stratification and perfect model semantics [Przmusinski 1988], stable model semantics [Gelfond and Lifschitz 1988; Przymusinski 1990], modularly stratified semantics [Ross 1994], well-founded semantics [Gelder et al. 1991] and alternating fixpoint semantics [Gelder 1993]; non-Horn clause logic programs such as disjunctive deductive databases [Fernández and Minker 1992a; Fernández and Minker 1992b; Barback 1992; Subrahmanian 1993], hypothetical reasoning [Bonner 1989; Bonner 1990; Chen 1997; Inoue 1994; Sattar and Goebel 1997]; constraints [Brodsky 1993; Jaffar and Maher 1994; Grumbach and Su 1996], etc. However, we believe that a separate survey is more appropriate for these issues. Indeed, the languages we survey here deal only with predicate stratification and fixpoint semantics.

Throughout this paper, we follow the Prolog convention and use words starting with upper-case letters for variables and "_" for anonymous variables.

2. COMPLEX-VALUE DEDUCTIVE LANGUAGES

The relational data model, which imposes the first normal form (1NF) restriction, has been found to be inexpressive for many novel database applications, such as engineering design, graphic databases, CAD/CAM, etc., as these applications require proper representation and manipulation of arbitrarily complex values with internal structures. The nested relational and complex-value data models [Abiteboul et al. 1987; Ozsoyogly and Yuan 1987; Roth et al. 1987; Roth 1988; Colby 1989; Levene and Loizou 1993; Abiteboul and Beeri 1995] developed during the past decade allow tuple components of a relation to be possibly nested tuples, sets or even relations.

Datalog is based on 1NF relations and therefore cannot support nested tuples and sets. In fact, its predecessor Prolog can indirectly support nested tuples by using functors. For example, we may use the following facts in Prolog to store information about cars and their owners:

car(*vehicle*(*ny_858778*, *chevy_vega*, 1983),
    owner(*name*(*arthur*, *keller*),
        getFrom(*dealer*)))

car(*vehicle*, (*ny_358735*, *volkswagen*, 1990)
  owner(*name*(*arthur*, *keller*),
    getFrom(*owner*(*name*(*kurt*, *godel*),
      getFrom(*owner*(*name*(*norbert*, *wiener*),
        getFrom(*dealer*)))))))

where *vehicle, name, owner, getFrom* are functors that are not interpreted (or are freely interpreted) because they have no prior meanings.

Prolog also indirectly supports sets by using lists. In Prolog, set expressions are represented using the built-in predicate *setof*:

$$setof(X, P, S)$$

where P represents a goal or goals, X is a variable occurring in P, and S is a list whose elements are sorted into a standard order with no duplicates and can be treated as a set. It is read as "The set of instances of X such that P is provable is S". In other words, $S = \{X | P\}$.

The problems with the *setof* predicate in Prolog are that it is based on lists and that it is not a first-order predicate so its semantics is not well-defined. Besides, the usage of lists as sets is not expressive enough. The simple membership predicate has to specify details about implementation, such as how to iterate over the lists. When a predicate involves more than one set, the program can become quite complicated and non-intuitive, which is contrary to the general philosophy of declarative programming [Kuper 1990].

For this reason, Datalog has been extended in the past several years to incorporate tuple and/or set constructors [Kuper 1990; Beeri 1991; Abiteboul and Grumbach 1991; Liu 1998b]. In the meantime, Prolog has also been extended to incorporate general-purpose set constructs and basic operations on sets [Dovier et al. 1996; Jayaraman 1992; Hill and Lloyd 1994].

In this section, we examine four different Datalog extensions: LDL, COL, Hilog and Relationlog.

2.1 LDL

LDL (Logical Data Language) [Tsur and Zaniolo 1986; Naqvi and Tsur 1989; Chimenti et al. 1990; Beeri et al. 1991] is the first language to extend Datalog to support complex values with well-defined semantics. It has been implemented at MCC. Like Prolog, LDL can indirectly support tuples by using functors. In addition, it directly supports sets.

Example 2.1 Consider the following nested relation *Dept_Employees*:[3]

Dept_Employees

Dept	*Employees*	
	Name	*Areas*
		Area
cpsc	*bob*	*db*
		ai
	joe	*os*
		pl
		db
math	*sam*	*gt*
		cm
		si
	tom	*ca*

We can represent this relation in LDL by using the following two facts:

$dept\_employees(\ cpsc,$
$\quad \{empl(bob, \{db, ai\}),$
$\quad empl(joe, \{os, pl, db\})\})$
$dept\_employees(\ math,$
$\quad \{empl(sam, \{gt, cm, si\}),$
$\quad empl(tom, \{ca\})\})$

To support access to elements in a set, LDL allows the use of the member predicate ($\in$). To support the construction of sets with rules, LDL provides two powerful mechanisms: set enumeration and set grouping. With set enumeration, a set is constructed by listing all of its elements. With set grouping, a set is constructed by defining its elements with a property that they must satisfy. Several examples are given below.

Example 2.2 To derive a relation on sets of book titles from the *book* base relation such that the total price of any three different books does not exceed \$100, we can use the following rule with a set enumeration term $\{X, Y, Z\}$ in LDL:

$$\begin{aligned}&book\_deal(\{X, Y, Z\}) \text{ :- } book(X, Px),\\ &\quad book(Y, Py), book(Z, Pz),\\ &\quad X \neq Z, Y \neq Z,\\ &\quad Px + Py + Pz < 100\end{aligned}$$

Note that we can specify the cardinality of the sets with set enumeration. This feature is not directly supported in Prolog.

Example 2.3 Consider the relation *parentof* represented as facts in LDL as follows:

$$parentof(bob, pam)$$
$$parentof(bob, tom)$$

The following set-grouping rule in LDL groups all parents of a person into a set and obtains $parentsof(bob, \{pam, tom\})$, where $\langle Y \rangle$ is a set-grouping term:

$$parentsof(X, \langle Y \rangle) \text{ :- } parentof(X, Y)$$

Unlike set enumeration, we cannot specify the cardinality of the sets with set grouping.

In LDL, programs must be predicate-stratified (or layered) based on grouping and negation in order to have a semantics. Given a rule of the form A :- L_1, ..., L_n, where $n \geq 0$, if L_i is negated or A has grouping, then this rule must be evaluated after the rules whose heads have the same predicate symbol as L_i, for each $1 \leq i \leq n$. That is, each L_i must be completely known when we evaluate this rule. Whether or not a program is predicate-stratified can be determined statically based on the predicate symbols in the program.

There are three significant limitations in LDL. First, the grouping mechanism is restricted to a single rule rather than to a set of rules (or a program).

Example 2.4 Consider the following facts:

$$fatherof(bob, pam)$$
$$motherof(bob, tom)$$

We cannot use the following grouping rules in LDL to group all parents of a person into a set and obtain $parentsof(bob, \{pam, tom\})$:

$$parentsof(X, \langle Y \rangle) \text{ :- } fatherof(X, Y)$$
$$parentsof(X, \langle Y \rangle) \text{ :- } motherof(X, Y)$$

as they actually generate $parentsof(bob, \{pam\})$ and $parentsof(bob, \{tom\})$ instead. We have to introduce an intermediate relation *parentof* and use it to obtain the desired result, as shown in Example 2.3.

The second problem is that grouping requires predicate stratification, which makes direct recursive definition of a nested relation impossible.

Example 2.5 Consider the following two rules:

$$ancestorsof(X, \langle Y \rangle) \text{ :- } parentsof(X, S), Y \in S$$
$$ancestorsof(X, \langle Y \rangle) \text{ :- } parentsof(X, S_1), Z \in S_1, ancestorsof(Z, S_2), Y \in S_2$$

Here we intend to define the nested relation *ancestorsof* directly and recursively. However, we cannot achieve what we intend in LDL as grouping is restricted to a single rule and grouping requires us to know the *parentsof* and *ancestorsof* relations in the body before we can evaluate the *ancestorsof* relation in the head in the second rule. One way around this problem is to introduce a flat intermediate relation *ancestorof* as follows:

$$ancestorof(X, Y) \text{ :- } parentsof(X, S), Y \in S$$
$$ancestorof(X, Y) \text{ :- } parentsof(X, S_1), Z \in S_1, ancestorof(Z, Y)$$
$$ancestorsof(X, \langle Y \rangle) \text{ :- } ancestorof(X, Y)$$

Finally, it is cumbersome to access deeply nested data and to nest/unnest relations, since we must be procedural by introducing intermediate variables and/or relations.

Example 2.6 Consider the LDL facts in Example 2.1 and two queries:

(1) Find the employee names in the computer science department for employees whose research areas include DB.

(2) Find if AI is a research area for employees.

We can represent them in LDL as follows by starting from the top level and gradually getting to the level we want and introducing intermediate variables S_1 and S_2:

$$? - dept\_employees(cpsc, S_1), empl(X, S_2) \in S_1, db \in S_2.$$

$$? - dept\_employees(\_, S_1), empl(\_, S_2) \in S_1, ai \in S_2$$

Example 2.7 The nested relation *Dept_Employees* in Example 2.1 can be obtained from the following normalized relation *Dept_Employee_Area* by using the nest operation of extended relational algebra defined in Roth et al. [1988] twice.

Dept_Employee_Area

Dept	*Employee*	*Area*
cpsc	*bob*	*db*
cpsc	*bob*	*ai*
cpsc	*joe*	*os*
cpsc	*joe*	*pl*
cpsc	*joe*	*db*
math	*sam*	*gt*
math	*sam*	*cm*
math	*sam*	*si*
math	*tom*	*ca*

In order to obtain *Dept_Employees* from *Dept_Employee_Area* in LDL, we have to use grouping several times and introduce several intermediate relations as follows:

$r1(D, E, \langle A \rangle)$:- $dept\_employee\_area(D, E, A)$

$r2(D, empl(E, As))$:- $r1(D, E, As)$

$dept\_employees(D, \langle Es \rangle)$:- $r2(D, Es)$

2.2 COL

COL (Complex Object Language) [Abiteboul and Grumbach 1991] is a typed extension of Datalog that supports complex values directly. Unlike LDL, which uses functor objects for tuples indirectly, COL directly supports tuples, tuple constructors and sets.

The nested relation *Dept_Employees* in Example 2.1 can be represented in COL as follows:

$dept\_employees(cpsc, \{[bob, \{db, ai\}], [joe, \{os, pl, db\}]\})$

$dept\_employees(math, \{[sam, \{gt, cm, si\}], [tom, \{ca\}]\})$

A novel feature of COL is the use of interpreted functors called data functions in a deductive framework. Data functions play a crucial role in functional data models and in many semantic data models. Their use in COL provides natural support for functional dependencies in deductive databases. In LDL, functional dependencies are not supported.

In COL, data functions and the member predicate ($\ni$) can be used to access sets and to construct sets through grouping or enumeration. Unlike LDL, where sets are unnamed, data functions in COL are used to name sets.

Consider the facts in Example 2.4 again. The *parentsof* relation can be defined in COL using the functor *par* as follows:

$par(X) \ni Y$:- $fatherof(X, Y)$

$par(X) \ni Y$:- $motherof(X, Y)$

$parentsof(X, par(X))$

The functor *par* is used to group every parent Y of a person X into a set denoted by $par(X)$ and grouping can involve more than one rule. Note that the unique set of parents associated with X is named by the data function $par(X)$.

In COL, we can group a set recursively using data functions.

Example 2.8 The task shown in Example 2.5 can be represented in COL as follows:

$anc(X) \ni Y$:- $par(X) \ni Y$

$anc(X) \ni Y$:- $par(X) \ni Z, anc(Z) \ni Y$

$ancestors(X, anc(X))$

The set enumeration of LDL shown in Example 2.2 can be performed in COL using a 0-ary functor f as follows:

$f \ni \{X, Y, Z\}$:- $book(X, Px), book(Y, Py), book(Z, Pz), X \neq Y, X \neq Z, Y \neq Z, Px + Py + Pz < 100$

$book\_deal(X)$:- $f \ni X$

As a result, COL allows grouping over several rules recursively with data functions. However, it is still cumbersome to access deeply nested data in order to nest/unnest relations.

To represent the two queries in Example 2.6 in COL, we still must introduce intermediate functions f and g as follows:

$$? - dept\_employees(cpsc, f),\ f \ni [X, g], g \ni db$$

$$? - dept\_employees(\_, f),\ f \ni [\_, g], g \ni ai$$

The following rules in COL show how to obtain the nested relation *Dept Employees* in Example 2.1 from the normalized relation *Dept_Employee_Area* in Example 2.7:

$f(D, E) \ni A$:- $dept\_employee\_area(D, E, A)$

$r1(D, E, f(D, E))$

$g(D) \ni [E, As]$:- $r1(D, E, As)$

$dept\_employees(D, g(D))$

As in LDL, programs in COL must be stratified as well in order to have a well-defined semantics. Stratification is based on predicate and function symbols used in the program. Given a rule of the form A :- $L_1, \ldots, L_n$, $n \geq 0$, if L_i is negated or contains a data function, or A contains a data function, then this rule must be evaluated after the rules whose heads have the same predicate or function symbol as L_i or A, for each $1 \leq i \leq n$. As in LDL, whether or not a program is stratified can be determined statically based on the predicate and function symbols occurring in the program.

For the task shown in Example 2.5, we might like to use the following two rules instead of the rules in Example 2.8 in COL:

```
anc(X) ∋ Y :-
    parents(X, S1),
    S1 ∋ Z,
    ancestors(Z, S2),
    S2 ∋ Y.

ancestors(X, anc(X)).
```

However, they are not stratified and therefore have no semantics in COL.

The stratification of COL has the following limitation:

Example 2.9 [Adapted from Kifer and Wu [1993]] Consider the following program in COL:

$person(peter, \{bridge\})$
$person(tom, \{chess, tennis\})$
$person(tom, hobby(peter))$

$hobby(X) \ni Y$:- $person(X, S), S \ni Y$

The intended meaning of these rules is that Peter's hobby is bridge and Tom's hobbies include chess and tennis and every hobby of Peter's is also a hobby of Tom's. However, the program is not stratified and therefore has no meaning.

2.3 Hilog

Hilog [Chen and Chu 1989; Chen and Kambayashi 1991] is a typed extension of Datalog intended to solve some of the problems of LDL. In Hilog, every set or tuple must be associated with a name that can be viewed as an atom or a functor of Prolog. The relation *Dept_ Employees* in Example 2.1 can be represented in Hilog by using one fact as follows:

```
dept_employees{
    dept(cpsc,
        employees{employee(bob,
                          areas{db, ai}),
                  employee(joe,
                          areas{os, pl, db})}),
    dept(math,
        employees{employee(sam,
                          areas{gt, cm, si}),
                  employee(tom,
                          areas{ca})})}
```

Note that the names associated with tuples and sets must be unique (unique name assumption). In the above example, we cannot change the name *dept employees* to *employees* as the latter is used inside the relation.

Sets in Hilog have a special meaning. For example, $areas\{db, ai\}$ does not mean that the set consists of db and ai. Instead, it means that db and ai are elements of the set, and is equivalent to $areas\{db\}$ and $areas\{ai\}$. With this special meaning, we can directly access deeply nested data either by giving their nested structure or using the unique name associated with them. The two queries in Example 2.6 can then be represented in Hilog directly as follows without introducing any intermediate symbols:

$? - dept(cpsc, employees\{employee(X, areas\{db\})\})$

$? - dept\_employees\{$
$\quad dept(\_, employees\{employee(\_, areas\{ai\})\})\}$

With the unique-name assumption, nested terms in Hilog can be directly used as atoms. Therefore, the second query above can be simply represented as follows:

$$? - areas\{ai\}$$

The nested relation *Dept_Employees* in Example 2.1 can be obtained from the normalized relation *Dept_Employee_Area* in Example 2.7 by using just the rule:

$dept(X, employees\{employee(Y, areas\{Z\})\})$:-
$\quad dept\_employee\_area(X, Y, Z)$

This set mechanism naturally supports grouping that involves several rules. The *parentsof* relation can be expressed in Hilog as follows:

$parentsof(X, parents\{Y\})$:- $fatherof(X, Y)$

$parentsof(X, parents\{Y\})$:- $motherof(X, Y)$

This set mechanism does not require stratification. The task shown in Example 2.5 can be represented in Hilog as follows:

$ancestorsof(X, ans\{Y\})$:- $parentsof(X, par\{Y\})$

$ancestorsof(X, ans\{Y\})$:- $parentsof(X, par\{Z\})$,
$\quad ancestorsof(Z, ans\{Y\})$

The unstratified COL program in Example 2.9 can be expressed in Hilog as follows:

$person(peter, hobbies\{bridge\})$

$person(tom, hobbies\{chess, tennis\})$

$person(tom, hobbies\{X\})$:-
$\quad person(peter, hobbies\{X\})$

However, the special treatment of sets in Hilog has its limitations since it does not allow us to specify explicitly what a set exactly contains. Therefore, the set enumeration of LDL is not supported. Another problem with Hilog is that the unique-name assumption is not practical for large databases.

The Hilog language is not higher-order in the sense that we cannot use variables for the names associated with the sets and tuples. In Section 4.1, we discuss a true higher-order language that has a similar name: HiLog.

2.4 Relationlog

Relationlog (Relation LOGic) [Liu 1995; 1998b] is another typed extension of Datalog with powerful set and tuple constructors. It has been implemented at the University of Regina [Liu and Shan 1998; Shan and Liu 1998]. Relationlog combines the best features of LDL, COL and Hilog. It directly supports complex values as COL. The nested relation *Dept_Employees* in Example 2.1 can be represented in Relationlog in the same way as in COL.

Relationlog allows negation, supports Hilog's special set treatment with partial set terms, and eliminates the unique-name assumption. A partial set term has the form $\langle X, Y, Z\rangle$ and corresponds to a Hilog set term $p\{X, Y, Z\}$ for some name p. It also supports set enumeration in LDL with complete set terms. A complete set term is of the form $\{X, Y, Z\}$ and corresponds to an LDL set-enumeration term of the same form.

Unlike set-grouping terms in LDL, partial set terms in Relationlog can appear not only in the head but also in the body of a rule. When a partial set term appears in the body, it denotes part of a set, as in Hilog. When appearing in the head, it is used to group a set as in LDL. Several rules can be used to group the same set.

Consider Example 2.4 again. The *parentsof* relation can be directly defined in Relationlog as follows:

$$parentsof(X, \langle Y \rangle) \text{ :- } fatherof(X, Y)$$
$$parentsof(X, \langle Y \rangle) \text{ :- } motherof(X, Y)$$

Here two rules are used to group the same set.

As in LDL and COL, programs must be stratified in order to have a semantics. Stratification is based on predicate symbols used in the program. Given a rule of the form A :- $L_1, \ldots, L_n$ where $n \geq 0$, if L_i is negated or contains a complete set term, then this rule must be evaluated after the rules whose head has the same predicate symbol as L_i, for each $1 \leq i \leq n$. As in LDL and also COL, whether or not a program is stratified can be determined statically based on the predicate symbols occurring in the program.

As the use of partial set terms in the head does not require stratification, the *ancestorsof* relation in Example 2.5 can be defined recursively in Relationlog as follows:

$$ancestorsof(X, \langle Y \rangle) \text{ :- } parentsof(X, \langle Y \rangle)$$
$$ancestorsof(X, \langle Y \rangle) \text{ :- } parentsof(X, \langle Z \rangle), ancestorsof(Z, \langle Y \rangle)$$

As in Hilog, deeply nested data can be directly accessed in Relationlog with partial set terms. The two queries in Example 2.6 can be represented in Relationlog directly as follows, without introducing any intermediate symbols:

$$? - dept\_employees(cpsc, \langle [X, \langle db \rangle] \rangle)$$
$$? - dept\_employees(\_, \langle [\_, \langle ai \rangle] \rangle)$$

We can obtain the nested relation *Dept_Employees* in Example 2.1 from the normalized relation *Dept_Employee_Area* in Example 2.7 by using only one rule, as in Hilog:

$$dept\_employee(X, \langle [Y, \langle Z \rangle] \rangle) \text{ :- } dept\_employee\_area(X, Y, Z)$$

The unstratified COL program in Example 2.9 can be expressed clearly as a stratified program in Relationlog as follows:

$$person(peter, \{bridge\})$$
$$person(tom, \langle chess, tennis \rangle)$$
$$person(tom, \langle X \rangle) \text{ :- } person(peter, \langle X \rangle)$$

The first fact says that Peter has exactly one hobby, which is bridge. The second fact says that Tom's hobbies include chess and tennis. The rule states that every hobby of Peter's is also a hobby of Tom's.

Relationlog has the following limitations. First, while it supports incomplete sets by using partial set terms, it does not support incomplete tuples (i.e., tuples with null values), which are common in database applications. For example, the tuple $fatherof(tom, \perp)$, representing that Tom's father is unknown, cannot be represented in Relationlog. Therefore, it is not clear what the meanings of recursive rules are when both incomplete sets and tuples appear. Next, since the complete set terms in Relationlog cannot contain partial set terms, it is not possible to have a set of which we know the cardinality but for which we do not have complete knowledge of some of the elements, such as $\{\langle a, b \rangle, \{b, c\}\}$. Besides, there is no easy way to query sets of a particular cardinality in Relationlog, such as sets of cardinality ≥ 2 or ≤ 2. Finally, being a complex-value language, Relationlog is lacking in its data-modeling power, as we discuss in the next section.

3. OBJECT-ORIENTED DEDUCTIVE LANGUAGES

Object-oriented concepts have evolved in three different disciplines: first in programming languages, then in artificial intelligence, and then in databases (since the end of the 60's). Indeed, object orientation offers some of the most promising ways to meet the demands of many advanced database applications. In this section, we first introduce the fundamental principles of the object-oriented data models and then examine their applications in deductive database languages.

Object Identity In a value-oriented database, keys, which are values, are used to represent objects in the real world. A fundamental problem with such a representation of objects is that keys are subject to updates. Updating a key causes a break in the continuity in the representation of the object. Furthermore, care must be taken to update all tuples and sets that refer to this object to reflect the change. Object-oriented databases permit the explicit representation of real-world objects through the use of object identifiers, which are different from values in the database. A unique object identifier within the entire database is assigned to each object that can be represented in the database and this association between object identifier and object remain fixed. Unlike values, an object identifier can be created or destroyed but cannot be updated. Two objects are different if they have different object identifiers even if they have the same attribute values. Whether or not the user can access object identifiers varies from system to system.

Complex Values Each object is associated with a value, which may be complex. The value associated with the object can contain object identifiers so that two objects can share an object by referencing the same object identifier. Updates to the values of the shared object do not affect the objects that refer to it. The value part of an object is also called its state [Beeri 1989; Kim 1990a].

Objects Since an object in the real world is associated with an object identifier and a value in an object-oriented database, a natural question is what an object is in a database. There are two kinds of views in object-oriented databases. One view is that everything is an object; that is, classes, object identifiers, atomic values, tuples, and sets are all objects. Objects are related to each other through attributes. The attribute values of an object can change but not the object itself. With this view, attributes of objects can be expressed in terms of functions and all attribute values together form a tuple. This view is used in Iris [Fishman et al. 1987] and Jasmine [Ishikawa 1993]. This view naturally incorporates functional and multivalued dependencies. The other view is that the pair of object identifier and its associated value together is an object. This view is used in the object-oriented data models O_2 [Lecluse and Richard 1989], Orion [Kim 1990a; 1990b] and ODMG-93 [Cattell 1996]. The first view results in a simpler semantics, which is very important for a logic-based language, while the second view allows object identifiers to identify not only tuples but also atomic values and sets.

Methods Objects in an object-oriented database are manipulated with methods. A method has a name, a signature, and an implementation. The name and signature provide an external interface to the method. The implementation is typically written in an extended programming language.

Classes A class is a set of objects that have exactly the same internal structure and therefore the same attributes and the same methods. The class to which an object belongs is called the immediate, proper or primary class of the object. A class directly captures the instance-of relationship between an object and the class to which it belongs. In addition, it provides the basis on which a query may be formulated. It also defines the attributes and methods of its instances. It is sometimes useful to allow an object to belong to more than one class. However, in most systems, an object must belong to only one class for performance reasons. In this case, an object is said to have a unique primary class. As we show shortly, inheritance makes it possible for an object to belong logically to more than one class.

Class Hierarchies and Inheritance Classes are organized into class hierarchies, which capture the generalization relationship between a class and its subclasses. Subclasses inherit the attributes and methods of their superclasses. An instance of a subclass logically belongs to its superclasses. Inheritance enables us to reduce the redundancy of the specification while maintaining its completeness. Inheritance normally takes place between classes, whereas instances are completely "sterile" in the object-oriented data models. A subclass can modify the attributes/methods inherited. This modification is called overriding. In addition, some attributes/methods defined in a superclass may not be applicable to its subclass. In this case, a subclass must block (or cancel) their inheritance. Inheritance can be single or multiple. In the case of single inheritance, the subclass hierarchy forms a tree; in the case of multiple inheritance, the subclass hierarchy forms a DAG. Multiple inheritance is more elegant than single inheritance, but conflicts may arise when an attribute/method name is defined in two or more superclasses. Several approaches are used to resolve an ambiguity. One is user-specified priority, which relies on the user to define a priority among the super classes whose attributes/methods are inherited. Another one is explicit renaming, which requires that all attributes/methods inherited must have distinct names. Overriding can also be used to resolve ambiguities. Unfortunately, there is still no consensus for the underlying semantics of multiple inheritance.

Encapsulation Encapsulation derives from the notion of abstract data types in programming languages. In an object-oriented database, objects encapsulate data and methods to be applied on these data. Encapsulation requires that information about a given object can be manipulated only by means of the methods defined for the object. The user or application program cannot directly examine or modify the value or the methods associated with the object. Therefore, encapsulation provides an abstract interface to the object and achieves some kind of logical data independence. It is a very useful notion for data modeling.

3.1 O-Logic

The object-oriented approach was first applied to a deductive framework in O-logic (Object Logic) [Maier 1986], which was in turn based on the logic programming language LOGIN [Aït-Kachi and Nasr 1986]. It treats object identifiers and atomic values as objects that can be related to each other through attributes. Classes can be used but are not objects.

Example 3.1 The following is an O-logic program:

$$\begin{aligned} &joe : employee[name \rightarrow \text{“}Joe\text{”},\\ &\qquad position \rightarrow prof,\\ &\qquad works \rightarrow cs]\end{aligned}$$

$$\begin{aligned} &sam : employee[name \rightarrow \text{“}Sam\text{”},\\ &\qquad position \rightarrow inst,\\ &\qquad works \rightarrow cs]\end{aligned}$$

$$\begin{aligned} &cs : dept[name \rightarrow \text{“}ComputerScience\text{”},\\ &\qquad head \rightarrow joe]\end{aligned}$$

$$\begin{aligned} &prof : rank[name \rightarrow \text{“}Professor\text{”},\\ &\qquad pay \rightarrow 2000]\end{aligned}$$

$$\begin{aligned} &inst : rank[name \rightarrow \text{“}Instructor\text{”},\\ &\qquad pay \rightarrow 1000]\end{aligned}$$

$$\begin{aligned} &E : employee[salary \rightarrow S] \text{ :-}\\ &\qquad E : employee[position \rightarrow R],\\ &\qquad R : rank[pay \rightarrow S]\end{aligned}$$

where *employee*, *dept*, *rank* are classes, *joe*, *sam*, *cs*, *prof*, *inst* are object identifiers, and *name*, *position*, *works*, *head*, *pay*, *salary* are attributes. The facts provide information about employees, departments, and pay rate of professors and instructors. The rule specifies how to infer the salary of each employee.

In O-logic, tuples and sets are not allowed. The objects can have arbitrary attributes but no structural or behavioral information can be defined or declared on their classes and therefore no inheritance is supported.

As rules can be used, a fundamental problem is how to generate objects with rules. This issue is not properly addressed in O-logic.

Example 3.2 Consider the following rule in O-logic:

$$
\begin{aligned}
&P : interesting\_pair[employee \rightarrow E,\\
&\qquad\qquad manager \rightarrow M] \text{ :-}\\
&\quad E : employee[name \rightarrow N, works \rightarrow D]\\
&\quad D : dept[manager \rightarrow M :\\
&\qquad\qquad employee[name \rightarrow N]]
\end{aligned}
$$

where P, E, M, D, and N are variables, and *interesting_pair*, *employee*, and *dept* are classes. The intended meaning of this rule is that if the employee's department's manager's name coincides with the employee's name, then the pair of employee and manager is interesting and an object identifier should be created for this pair by using the variable P that occurs only in the head.

The problem with the rule in Example 3.2 is that it is not clear how the variable P should be quantified [Kifer and Wu 1993]. Universal quantification over P obviously does not make sense. It is suggested in Maier [1986] that P should be existentially quantified as $(\forall E)(\forall M)(\forall N)(\exists P)$. However, the argument for such a quantification is not well substantiated. It is pointed out in Kifer and Wu [1993] that the correct quantification should be $(\forall E)(\forall M)(\exists P)(\forall N)$. Both happen to work here because E and M functionally determine N. If the label *name* is set-valued, the two quantifications will yield different results. There is no obvious way of choosing between these two quantifications using only the syntactic structure of the rule. The problem is that the variable P does not appear in the rule body so that the rule is not domain-independent [Abiteboul 1995].

3.2 F-Logic

An extended O-logic based on O-logic was first proposed in Kifer and Wu [1993], followed by a more general F-logic (Frame Logic) in Kifer and Lausen [1989] and Kifer et al. [1995] as a solution to the problems of O-logic. F-logic has been partially implemented at Universität Freiburg in Germany [Frohn et al. 1997]. It treats atomic values, object identifiers, functor objects, even attribute and classes as objects. It allows the declaration of attributes on classes and supports attribute inheritance.

Example 3.3 The following is an example of an F-logic program:

$$
\begin{aligned}
&person[name \Rightarrow string,\\
&\qquad age \Rightarrow integer,\\
&\qquad parents \Rrightarrow person]\\
&employee :: person\\
&joe : employee\\
&tom : person\\
&pam : person\\
&25 : integer\\
&\text{“}Joe\text{”} : string\\
&joe[name \rightarrow \text{“}Joe\text{”},\\
&\quad age \rightarrow 25,\\
&\quad parents \twoheadrightarrow \{tom, pam\}]
\end{aligned}
$$

where *person*, *employee*, *integer*, and *string* are classes, while *joe*, *tom*, *pam*, 25 and *“Joe”* are object identifiers. The symbols “::” and “:” denote subclass relationship and class membership respectively. The statement *employee*::*person* says that *employee* is a subclass of *person*, *joe*:*employee* says that *joe* is an instance of the class *employee*, etc. Attributes applicable to the class *person* are declared by using double-shafted arrows $\Rightarrow$ and $\Rightarrow\!\!\!\Rightarrow$, where the former is for scalar attribute definitions and the latter for set-valued attribute definitions. They are inherited by the subclass *employee*. Attribute values of the object *joe* are represented by using single-shafted arrows $\rightarrow$ and $\twoheadrightarrow$, where the former is for scalar values and the latter for set values, corresponding to double-shafted arrows.

F-logic is a powerful deductive language with a well-defined semantics compared to O-logic. It supports functor objects and sets. It also allows the use of functor objects as classes, attributes, and object identifiers. The interesting-pair rule of O-logic in Example 3.2 can be represented in F-logic as follows:

$$\begin{array}{l} f(E, M) : interesting\_pair \\ \qquad [employee \rightarrow E, manager \rightarrow M] \text{ :- } \\ E : employee[name \rightarrow M : string, works \rightarrow D] \\ D : dept[manager \rightarrow M : employee[name \rightarrow N] \end{array}$$

where f is a functor and the rule infers functor objects such as $f(bob, sam)$ as instances of the class *interesting pair*.

This object-creation mechanism is also used in the object-oriented deductive languages C-logic [Chen and Warren 1989] and LIVING IN A LATTICE [Heuer and Sander 1993].

Unlike O-logic, which only supports simple attributes, F-logic supports parameterized attributes. For example, we can have the following program in F-logic:

$$\begin{array}{l} person[children\_with@person \Rightarrow\!\!\!\Rightarrow person] \\ bob[children\_with@liz \twoheadrightarrow \{tom, pam\}] \\ bob[children\_with@ann \twoheadrightarrow \{jim\}] \end{array}$$

F-logic has several drawbacks. First, there is no distinction between classes and objects, since the domains for classes and objects are the same. They can be distinguished only from their occurrences in the program. The attribute declaration information cannot really be strictly separated from attribute value information in F-logic. As a result, there is no clear separation between the notions of schema and instance, which is essential for database systems.

F-logic is a pure object-oriented deductive language. It does not directly support relationships as do in value-oriented deductive languages discussed in the previous section. It only supports monotonic multiple attribute inheritance, and does not support attribute overriding and blocking.

In F-logic, sets are not treated as objects and cannot have attributes. Functor objects cannot contain sets. As a result, the object-creation mechanism in F-logic is limited, since we cannot create objects directly based on some sets. Furthermore, the arguments associated with attributes cannot be sets either.

The treatment of sets in F-logic is the same as in Hilog. For a statement $bob[age \rightarrow 30]$, 30 is the value of the attribute *age* of *bob*. If a set is involved, the semantics is different. For a statement $bob[children \twoheadrightarrow \{jim, pat\}]$, the set $\{jim, pat\}$ does not mean that it is the value of the *children* of *bob*. Instead, it means it is part (subset) of the set value and therefore it is possible to have another statement such as $bob[children \twoheadrightarrow \{sam\}]$. The (complete) value of *children* of *bob* is the union of all such partial values that can be inferred from the program. This special treatment allows the complete value of a set-valued attribute to be inferred in a number of steps using several rules. The following is an example:

$$X[ancestors \twoheadrightarrow \{Y\}] \text{ :- } X[parents \twoheadrightarrow \{Y\}]$$

$$\begin{array}{l} X[ancestors \twoheadrightarrow \{Y\}] \text{ :- } X[parents \twoheadrightarrow \{Z\}], \\ \qquad\qquad\qquad\qquad\qquad Z[ancestors \twoheadrightarrow \{Y\}] \end{array}$$

As in Hilog, this unique treatment of sets makes it impossible to specify the complete value of a set-valued attribute. For example, if we know that *tom* and *pam* are the two parents of *joe*, we can only use the statement:

$$joe[parents \twoheadrightarrow \{tom, pam\}]$$

which will not prevent additional persons from being inferred as *joe*'s parent.

Next, F-logic is an untyped language with a complicated syntax and semantics full of various symbols. However, important features such as typing and unique primary class (the lowest class in the class hierarchy where the object is an instance of) for objects are not built into the semantics, but are dealt with using axioms. In terms of programming, this means that the user should take care of them by properly including corresponding axioms into the programs if he wants his program to be typed or objects to have unique primary class. As well, the notion of typing expressed in axioms can only be applied to attribute values rather than functor objects.

Finally, F-logic does not support behavioral object-oriented features such as methods and encapsulation.

3.3 ROL

ROL (Rule-based Object Language) [Liu 1996; 1998a] is a recently proposed object-oriented deductive language implemented at the University of Regina [Liu et al. 1998]. It is based on F-logic and solves some of the problems of F-logic discussed above.

Unlike F-logic, ROL is a typed language. Objects and classes are strictly distinguished and function at two different levels: schema and instance. At the schema level, we declare classes, attributes on classes, and subclasses. The attribute declarations of a class constrain the instances of the class. At the instance level, we use facts and rules to provide extensional and intensional information about objects. In addition, ROL allows both partial and complete information about set-valued attribute to be specified by supporting limited forms of partial and complete set terms of Relationlog.

The F-logic program in Example 3.3 can be expressed equivalently in ROL as follows:

Schema

$$\begin{aligned}&person[name \Rightarrow string,\\ &\qquad age \Rightarrow integer,\\ &\qquad parents \Rightarrow \{person\}]\\ &employee\ isa\ person\end{aligned}$$

Facts

$$\begin{aligned}&tom : person\\ &pam : person\\ &joe : employee[name \rightarrow \text{“Joe”},\\ &\qquad age \rightarrow 25,\\ &\qquad parents \rightarrow \langle tom, pam\rangle]\end{aligned}$$

where *person*, *employee*, *integer*, and *string* are classes, *joe*, *tom*, and *pam* are object identifiers, and $\langle tom, pam\rangle$ is partial set term. Unlike F-logic, ROL builds in value classes such as integer, string, real, etc. The symbols *isa* and : are used to denote immediate subclass relationship and primary class membership respectively. The double-shafted arrow $\Rightarrow$ is used for both scalar and set-valued attribute definitions and the single-shafted arrow $\rightarrow$ is used for both scalar and set-valued attribute values. If we want to represent that *tom* and *pam* are the only parents of *joe*, we can replace the partial set term $\langle tom, pam\rangle$ with the complete set term $\{tom, pam\}$.

Unlike F-logic, ROL has a simple syntax and semantics. The semantics builds-in important object-oriented features, such as typing, multiple inheritance and unique primary class. In ROL, functor objects are required to be typed with respect to their class definitions. For example, functor objects $interesting\_pair(bob, sam)$ and $list(1, list(2, list(3, nil)))$ are instances of the functor class *interesting_pair* and *list*(*integer*, *list*), respectively.

A novel feature of ROL is that its functor classes generalize classes and relations. Functor objects can directly represent relationships in ROL. They can have attributes and can be used as attribute values. In addition, functor objects can contain sets and thus are more general than those in F-logic. As a result, ROL subsumes predicate-stratified Datalog (with negation) and LDL without grouping and Relationlog without partial set terms as special cases. For example, the following LDL rules are also ROL rules:

$int(s(X))$:- $int(X)$

$ancestor(X, Y)$:- $parent(X, Y)$

$ancestor(X, Y)$:- $parent(X, Z)$,
$ancestor(Z, Y)$

$book\_deal(\{X, Y, Z\})$:- $book(X, Px)$,
$book(Y, Py)$, $book(Z, Pz)$
$X \neq Y, X \neq Z, Y \neq Z$,
$Px + Py + Pz < 100$

The interesting-pair rule in Example 3.2 can be represented in ROL as follows:

$interesting\_pair(E, M)$:-
$E : employee[name \rightarrow N, works \rightarrow D]$,
$D : dept[manager \rightarrow M]$,
$M : employee[name \rightarrow N]$

Here, a typed functor object *interesting_pair* (*E*, *M*) that has no separate identifier is used for each interesting-pair object, whereas in F-logic, an untyped functor object of the class *interesting_pair* is used as an identifier for the associated interesting pair.

ROL supports non-monotonic multiple attribute inheritance, as discussed in Borgida [1988], whereas F-logic supports only monotonic multiple attribute inheritance. Subclasses in ROL can inherit attribute declarations of their superclasses and can also override such inheritance.

Example 3.4 In an international courier service address database, we may have three classes *address*, *us_address* (for U.S.A.), *cn_address* (for Canada), so that *us_address* and *cn_address* are subclasses of *address*. We may define the attribute *postcode* of *address* on *integer* which also applies to instances of *us_address*. However, the *postcode* for *cn_address* must be defined on *string*. Therefore, *cn_address* should override the inherited attribute declaration of *postcode* from *address*. This can be achieved in ROL with the following class definitions, but is not supported in F-logic:

$address[..., postcode \Rightarrow integer]$

$us\_address\ isa\ address$

$cn\_address\ isa\ address$
$[postcode \Rightarrow string]$

In ROL, a subclass can also block the inheritance from its superclasses to itself and to its subclasses by using the built-in class *none*.

Example 3.5 Consider the classes *person*, *french*, *orphan*, and *french_orphan* where *french*, and *orphan* can be defined as subclasses of *person* and *french_orphan* as a subclass of both *french* and *orphan*. If we declare attributes *father*, *mother*, and *age* for *person*, then it is meaningful for *french* to inherit or override all attribute declarations from *person*, but not meaningful for *orphan* and *french_orphan* to inherit *father* and *mother* attributes. That is, *orphan* and *french_orphan* should block the inheritance of attribute *father* and *mother* from *person*. This intended semantics is naturally supported in ROL using the following class definitions:

$person[father \Rightarrow person$,
$mother \Rightarrow person$,
$age \Rightarrow integer]$

$orphan\ isa\ person$
$[father \Rightarrow none,$
$mother \Rightarrow none]$

$french\ isa\ person$

$french\_orphan\ isa\ french, orphan$

Even if *french* overrides (more accurately, refines) the attributes *father* and *mother* as follows:

$french\ isa\ person$
$[father \Rightarrow french,$
$mother \Rightarrow french]$

the attribute *father* and *mother* are still blocked in *french_orphan* because of the use of *none*. If we insist that *french_orphan* have mother *nun*, then we can use the following class definition to override the inheritance blocking:

$french\_orphan\ isa\ french, orphan$
$[mother \Rightarrow nun]$

Another novel feature of ROL is that sets are treated as first-class citizens as values, object identifiers, and functor objects, so that sets can also have attribute. For example, we can have the following class definition and fact in ROL:

$\{person\}[count \Rightarrow integer]$

$\{tom, pam\}[count \rightarrow 2]$

ROL has the following limitations. First, like F-logic, an object identifier in ROL can directly identify only a tuple but not a set or an atomic value. ROL's treatment of classes is not general enough as there are only two layers, objects and classes, and classes cannot be treated as objects (or meta objects). In ConceptBase [Jarke et al. 1995], there can be infinite layers of objects: ordinary objects, classes, metaclasses, etc. Besides, parameterized attributes of F-logic are not supported in ROL. Another problem with ROL is that different classes not related by the subclass relationship cannot have attributes of the same name.

Although ROL is a deductive language that supports both object-oriented and value-oriented approaches, its support for the value-oriented approach is limited as it does not support grouping in functor objects. Its partial and complete set mechanisms are not as general as those in Relationlog.

Finally, behavioral object-oriented features such as methods and encapsulation are not supported in ROL.

3.4 IQL

IQL (Identity Query Language) [Abiteboul and Kanellakis 1989] is a typed deductive database language based on the object-oriented data model O_2. As in O_2, an object in IQL is a pair of object identifier and value. Unlike F-logic and ROL, the value in an IQL object can be not only a tuple, but also a set or an atomic value. For example, we can have the following objects for a family in IQL:

$(o_1, [name : \text{“}Bob\text{”}, spouse : o_2,$
$children : o_5, phone : o_6])$

$(o_2, [name : \text{“}Liz\text{’}, spouse : o_1,$
$children : o_5, phone : o_6$

$(o_3, [name : \text{“}Tom\text{”}, phone : o_6])$

$(o_4, [name : \text{“}Pam\text{”}, phone : o_6])$

$(o_5, \{o_3, o_4\})$

$(o_6, 1234567)$

In this example, object identifiers o_1, o_2, o_3 and o_4 identify tuples, o_5 identifies a set, while o_6 identifies an atomic value. If this family changes its phone number or add another child, only one simple update is needed.

In IQL, the value associated with an object identifier can be accessed by using the object identifier dereferencing operator ^. For example, $\widehat{o_1}$, $\widehat{o_5}$ denote the tuple $[name : \text{“}Bob\text{”}, spouse : o_2,$

$children : o_5, phone : o_6]$ and the set $\{o_3, o_4\}$, respectively.

IQL directly supports relations in addition to classes. In pure object-oriented data models such as O_2 and ODMG-93, many-to-many relationships between objects are difficult to deal with. But they can be represented in IQL by using relations. Indeed, using object identifiers for every object is burdensome and problematic in object-oriented deductive languages, and a pure value-oriented approach has been argued to be better in this regard [Ullman 1991]. With direct support for relations, IQL combines the benefits of the relational and object-oriented approaches.

In F-logic and ROL, object identifiers are simply constants that can be introduced explicitly in facts and rules by the user. In IQL, object identifiers are system-created, as in object-oriented programming languages. They can be created only by using rules that have variables occurring in the head, but not in the body as in O-logic. This may be viewed as a syntactic variation of the *new* construct in object-oriented programming languages. Unlike O-logic, there is a well-defined semantics for this object-creation mechanism.

The intended *interesting_pair* rule in Example 3.2 can be represented in IQL as follows:

$pair(E, M)$:-
  $employee(E, N, D)$,
  $dept(D, M)$,
  $employee(M, N, D)$

$interesting\_pair(O, E, M)$:- $pair(E, M)$

$\hat{O} = [E, M]$:- $interesting\_pair(O, E, M)$

The first rule generates all pairs in the relation *pair*. The second creates an object identifier for each pair with the relation *interesting_pair* using the variable O that appears in the head but not in the body. The last rule assigns the pair to the object identified by the corresponding object identifier.

Object identity in IQL can be used not only for object sharing and update management, but also for set grouping as data function of COL.

Example 3.6 The following rules show how to derive the relation *ancestorsof* from the relation *parentsof* using object identifiers as grouping in IQL.

$temp(X, O)$:- $parentsof(X, Y)$

$\hat{O}(Y)$:- $temp(X, O), parentsof(X, Y)$

$\hat{O}(Y)$:- $temp(X, O), \hat{O}(Z), parentsof(Z, Y)$

$ancestorsof(X, \hat{O})$:- $temp(X, O)$

The first rule is used to create an object identifier for the set of ancestors for each person. The next two rules use the object identifiers to group all ancestors for each person. The last rule generates the relation *ancestorsof* by dereferencing object identifiers that identify sets.

Note that the rules in the above example must be stratified; that is, the last rule should be evaluated after the first three rules. Unlike Datalog (with negation), LDL, COL and Relationlog, in which stratification is automatically determined based on predicate or function symbols, in IQL we may not have such symbols to use, as shown in the second and third rule in Example 3.6. Therefore, the user of IQL must take care of stratification by using program composition. The rules in Example 3.6 should be organized into two programs with the first three rules in the first one and the last in the second.

IQL's object identifier-creation mechanism is also used in object-oriented deductive languages LOGRES [Cacaceet al. 1990] and LLO [Lou and Ozsoyoglu 1991].

Like ROL, IQL is a typed language. There is a clear separation of the notions of instances and schema. Objects and classes are strictly distinguished and function at two different levels: instance and schema. At the schema level, we declare classes and their associated types. The kinds of object identifiers that can be created with rules are completely determined by the types associated with the classes for these objects.

IQL has the following problems: First, the object identifier-creation mechanism may lead to infinite loops and does not support a logic-oriented semantics, as discussed in Beeri [1990]. Indeed, two kinds of semantics are used to define IQL, one for the underlying data model and one for the IQL language, which makes the language quite complicated and nonintuitive.

Second, IQL is so dependent on relations that no objects can exist if there are no relations. Therefore, it is not a pure object-oriented deductive database language.

Another problem is that values such as integers, strings, tuples, sets, etc., cannot be treated as objects that have properties in IQL. They can appear only in the value part of an object. For a mathematical database, we may store integers and their associated attributes such as odd, even, prime, etc. For example, $(2, [even : true, prime : true])$. In this case, an integer is identified by itself rather than by a separate identifier. The cases for tuples and sets are the same.

In addition, separating relations from classes suffers the same problems associated with the entity-relationship model. For example, relationships can only include classes, while classes cannot include relationships and relationships cannot include relationships.

Finally, the schema supported by IQL is quite limited. It does not support important object-oriented features such as attribute inheritance or behavioral object-oriented features such as methods and encapsulation. Indeed, how to incorporate methods and encapsulation in an object-oriented deductive framework remains an open problem.

4. SCHEMA AND HIGHER-ORDER FEATURES

A database usually has a schema that provides the description of the database structure, constrains data in the database and query results, and guarantees part of the consistency of the database. In relational database systems, schematic information is also stored in relations and the same relational languages can be used to query not only data but also the schema.

However, the notion of schema is not properly supported in deductive database languages, because they are based on the logic programming language Prolog, which is inherently not typed. Indeed, the semantics of schema is not well defined in most deductive frameworks.

In Datalog, predicates correspond to relation names. In order to reason about schematic information, we need to use predicate variables that belong to higher-order logic. However, higher-order logics have been met with skepticism, since the unification problem is undecidable. A non-logic-based language has been used to specify the schematic information in a few implemented deductive database systems such as LDL, and such information cannot be queried, let alone by using a logic-based language.

Prolog actually combines first-order logic, higher-order constructs, and meta-level programming in one working system, so that generic predicates such as transitive closure and sorting can be defined, predicates can be passed as parameters and returned as values, and predicates and rules can be queried.

Example 4.1 Prolog provides a built-in binary predicate *current_predicate* $(Name, Term)$ that can be used to unify $Name$ with the name of a user-defined predicate, and $Term$ with the most general term corresponding to that predicate. Consider the following rules in Prolog:

relation(X) :- *current_predicate*(X, Y)

structure(Y) :- *current_predicate*(X, Y)

The first rule defines a relation to collect all predicates (relation names) in the database. The second rule defines a relation to collect all predicate structures (predicates and their arguments) in the database.

Unfortunately, the semantics of Prolog is based on first-order logic, which does not have the wherewithal to support any of these features. Therefore, they have an ad hoc status in logic programming.

In this section, we examine four higher-order languages that can be used to reason about schematic information in deductive databases: HiLog, L^2, F-logic and ROL. The first two are value oriented, while the last two are object oriented.

4.1 HiLog

In first-order logic, predicates and functions have different roles. Predicates are used to construct atoms, while functions are used to construct terms that are embedded in other terms or atoms. Predicate and function symbols have fixed arities. However, it has been shown from Prolog applications that it is very useful to let a symbol be used as both predicate and function with various arities.

HiLog (Higher-order Logic) [Chen et al. 1993] generalizes such a usage and gives it a well-defined declarative semantics. Note that HiLog here is different from Hilog discussed in Section 2.3. There are only two kinds of symbols in HiLog, parameters with various arities (or arityless) and variables. Parameters can function as predicates, functions and constants depending on the context. Terms are formed with parameters and variables in the usual way and may represent individuals, terms, and atoms in different contexts.

In first-order logic, unary relations can be viewed as sets intensionally. In HiLog, unary parameterized terms can be viewed as sets in the same way. For example, part of the relation in Example 2.1 can be represented in HiLog as follows:

areas(*bob*)(*db*)
areas(*bob*)(*ai*)
areas(*joe*)(*os*)
areas(*joe*)(*pl*)
areas(*joe*)(*db*)

employees(*cpsc*)(*bob*, *areas*(*bob*))
employees(*cpsc*)(*joe*, *areas*(*joe*))

dept_employee(*cpsc*, *employees*(*cpsc*))

As HiLog does not support extensionally represented sets, we do not classify it as a complex-value deductive language. However, variables can be used in place of predicates in HiLog, so that we can use such variables to reason about schematic information.

Example 4.2 Consider the database consisting of three relations: *hardware, software*, and *admin*, each of which represents the employees in the respective departments. The tuples in all three relations have attributes *name, age* and *salary*. We can define the following relations with higher-order queries in HiLog:

$p_1(R)$:- $R(\_, \_, \_)$

$p_2(R)$:- $R(joe, \_, \_)$

$p_3(N)$:- $R(N, \_, \_)$

$p_4(N)$:-
  $R_1(joe, A_1, \_), R_2(N, A_2, \_),$
  $A_1 < A_2$

$p_5(N_1, N_2)$:-
  $R_1(N_1, A, \_), R_2(N_2, A, \_),$
  $R_1 \neq R_2, N_1 \neq N_2$

$p_6(N_1, N_2)$:-
  $R_1(N_1, \_, S), R_2(N_2, \_, S),$
  $N_1 \neq N_2$

$p_7(N)$:-
  $R(N, \_, S), \neg R(N', \_, S')$,
  $S < S'$

The relation p_1 collects the relation names in the database. The relation p_2 collects the department names (relation names) to which *joe* belongs. The relation p_3 collects the names of employees in the *hardware*, *software* and *admin* relations. The relation p_4 collects the names of employees who are older than Joe. The relation p_5 collects the pair of the names of employees who have the same age but are in different departments. The relation p_6 collects the pair of the names of employees who have the same salary. The relation p_7 collects the names of employees who have the highest salary in their department.

However, we cannot define a relation to collect all relation names in HiLog since we must explicitly deal with the arity of each relation in order to query the relation name, as shown in the first rule in Example 4.2. Furthermore, we cannot query attributes as they are not representable in HiLog. Therefore, we cannot define in HiLog the two relations *relation* and *structure*, as in Example 4.1.

4.2 L^2

L^2 [Krishnamurthy and Naqvi 1988] is a higher-order deductive database language based on LDL. The problems discussed above are solved by introducing attributes into tuples and redefining the database as a tuple.

Example 4.3 The database discussed in Example 4.2 can be defined in L^2 as a tuple as follows:

$$db = (hardware : r_1,\ software : r_2,\ admin : r_3)$$

where *hardware*, *software* and *admin* are attribute names that function as predicates or relation names of the database tuple db, and r_1, r_2, r_3 are corresponding relations, each of which is a set of tuples of the form $(name : \ldots, age : \ldots, salary : \ldots)$. A specific attribute value *joe* in the relation *hardware* can be referenced with $.hardware(.name = joe)$. Besides $=$, we can also use $\neq$, $<$, $\leq$, $>$, and $\geq$.

In L^2, each relation has an ordering of attributes in the tuples, so that attributes can be omitted according to this ordering. We can use variables for attribute names (relation names) and we need not deal with their arity.

Example 4.4 Based on the database in Example 4.3, we can define the following relations with higher-order queries:

$.p_1(R)$:- $.R$

$.p_2(R)$:- $.R(.name = joe)$

$.p_3(N)$:- $.R(.name = N)$

$.p_4(N)$:-
  $.R_1(.name = joe, .age = A)$,
  $.R_2(.name = N, .age > A)$

$.p_5(N_1, N_2)$:-
  $.R_1(.name = N_1, .age = A)$,
  $.R_2(.name = N_2, .age = A)$,
  $R_1 \neq R_2$

$.p_6(N_1, N_2)$:-
  $.R_1(.name = N_1, .salary = S)$,
  $.R_2(.name = N_2, .salary = S)$,
  $N_1 \neq N_2$

$.p_7(N)$:-
  $.R(.name = N, .salary = S)$,
  $R \neg (.name \neq X, .salary > S)$,

$.p_8(A)$:- $.hardware(.A)$

$.p_9(R, \langle A \rangle)$:- $.R(.A)$

The relation p_1 collects all relation names in the database no matter what arity they have. It is the L^2 version of the first rule in Example 4.1. The relations p_2 to p_7 are the same as in Example 4.2. The relation p_8 collects all the attribute names of the relation *hardware* in the database. The relation p_9 collects all relation names and their attributes, where $\langle A \rangle$ is a set-grouping term. Its definition is more intuitive than the second rule in Example 4.1.

A novel feature of L^2 is the introduction of replacement semantics as a solution to higher-order unification. In bottom-up evaluation of rules in deductive databases, general unification is not needed. Instead, only matching is used; that is, only one of the two terms contains variables. The higher-order variables in L^2 are limited to range over database attributes. A rule with higher-order variables can be rewritten by replacing the variables with attributes. The rewritten rules are in first-order logic and their meaning is well defined. Since the number of database attributes must be finite, the language is decidable. This approach is a natural and simple way towards the integration of the definition and manipulation of schema and data.

However, L^2 does not have a notion of schema. In the database in Example 4.3, if one of the relations r_1, r_2, r_3 is empty, then we have no structural information such as *name*, *age*, and *salary*. Furthermore, it is not required that tuples in a relation have the same form. For example, r_1 can contain two tuples: $(name : joe, age : 30, salary : 30K)$ and $(name : sam, wife : pam)$. Therefore, the last two rules in Example 4.4 are not really meaningful in this case.

4.3 F-logic

In pure object-oriented databases, we have classes instead of relations. A class denotes a set of objects that have common attributes and prescribes the attributes applicable to their instances. Classes are organized into a subclass hierarchy such that the prescribed attributes of superclasses can be inherited by the subclasses.

A natural question for object-oriented databases is how to reason about the subclass hierarchy, attribute information of classes, and class and attribute information of objects. F-logic is the first object-oriented deductive language that supports reasoning about class and attribute information.

Example 4.5 The following is a portion of an F-logic program based on the database discussed in Example 4.1. It has classes and subclass definitions and part of the objects.

$$employee :: person$$

$$hardware :: employee$$

$$software :: employee$$

$$admin :: employee$$

$$joe : hardware$$

$$\begin{aligned} employee[&name \Rightarrow string,\\ &age \Rightarrow integer,\\ &salary \Rightarrow integer,\\ &children \Rightarrow\!\!\!\!\!\Rightarrow person] \end{aligned}$$

$$\begin{aligned} joe[&name \rightarrow \text{“}Joe\text{”},\\ &age \rightarrow 30,\\ &salary \rightarrow 30K,\\ &children \twoheadrightarrow \{bob, sam\}] \end{aligned}$$

$$\ldots$$

Unlike L^2, we can define attribute information for classes in F-logic. Even though we have no instances in the class, the attribute information remains.

As discussed in Section 3, classes and attributes are also objects in F-logic. They differ from other objects in their occurrences in special places. Therefore, we can use variables for the places of classes and attributes and reason about class and attribute information in F-logic.

Example 4.6 For the F-logic program in Example 4.5, we can use the following rules with higher-order queries to collect all classes, all attributes of the class *hardware*, all attributes of each class, and all instances of each class in F-logic:

$db[classes \twoheadrightarrow \{C\}]$:- $C :: D$
$db[classes \twoheadrightarrow \{C\}]$:- $D :: C$
$db[classes \twoheadrightarrow \{C\}]$:- $O : C$
$db[classes \twoheadrightarrow \{C\}]$:- $C[A \Rightarrow D]$
$db[classes \twoheadrightarrow \{C\}]$:- $C[A \Rrightarrow D]$
$db[classes \twoheadrightarrow \{C\}]$:- $D[A \Rightarrow C]$
$db[classes \twoheadrightarrow \{C\}]$:- $D[A \Rrightarrow C]$

$employee[subclasses \twoheadrightarrow \{C\}]$:-
  $C :: employee$

$hardware[attributes \twoheadrightarrow \{A\}]$:-
  $hardware[A \Rightarrow C]$

$hardware[attributes \twoheadrightarrow \{A\}]$:-
  $hardware[A \Rrightarrow C]$

$C[attributes \twoheadrightarrow \{A\}]$:- $C[A \Rightarrow D]$

$C[attributes \twoheadrightarrow \{A\}]$:- $C[A \Rrightarrow D]$

$C[instances \twoheadrightarrow \{O\}]$:- $O : C$

As shown in the above example, it is cumbersome to query all classes in F-logic. The reason is that F-logic does not distinguish between ordinary objects and classes. As a result, we have to query all possible places for classes. The use of two kinds of arrows is somewhat burdensome, too. Similarly, it is also cumbersome to query all immediate subclasses, superclasses, and immediate instances of a class in F-logic. For example, the following rules show how to collect all immediate subclasses[1]

$C[immediate\_subclasses \twoheadrightarrow \{D\}]$:-
  $C :: D$,
  $\neg C[intervening\_class@D \twoheadrightarrow \{E\}]$

[1]This example is due to one of the referees.

$C[intervening\_class@D \twoheadrightarrow \{E\}]$:-
  $C :: E, E :: D$,
  $C \neq E, E \neq D$

4.4 ROL

ROL [Liu 1996; 1998a] is another language that supports reasoning about subclass hierarchy, attribute information of classes, and class and attribute information of objects. It has a clear notion of schema and allows class and attribute variables in rules. The semantics of higher-order features of ROL is based on replacement semantics of L^2 so that the language is still decidable.

Example 4.7 The F-logic program in Example 3.3 can be represented in ROL as follows:

Schema
  person
  employee isa person
    $[name \Rightarrow string$,
    $children \Rightarrow \{person\}$,
    $phone \Rightarrow integer]$
  hardware isa employee
  software isa employee
  admin isa employee

Facts
  $joe : hardware$
   $[name \rightarrow$ "*Joe*",
   age $\rightarrow$ 30,
   $salary \rightarrow 30K$,
   $children \rightarrow \{bob, sam\}]$
  . . .

In ROL, the user explicitly defines immediate subclasses and primary class membership using the symbols "*isa*" and ":", respectively, based on which general subclasses and class membership can be derived using the symbols "*isa**" and ":*", respectively.

Example 4.8 Consider the following ROL rules with higher-order queries based on the ROL program in Example 4.7:

$db[classes \rightarrow \langle C \rangle]$:- C

$employee[subclasses \rightarrow \langle C \rangle]$:-
    C $isa*$ $employee$

$hardware[attributes \rightarrow \langle A \rangle]$:-
    $hardware[A \Rightarrow \_]$

$C[attributes \rightarrow \langle A \rangle]$:- $C[A \Rightarrow \_]$

$C[immediate\_subclasses \rightarrow \langle D \rangle]$:-
    D isa C

$C[non\_immediate\_subclasses \rightarrow \langle D \rangle]$:-
    D $isa*$ C, $\neg$ D isa C

$C[immediate\_instances \rightarrow \langle O \rangle]$:- $O : C$

$C[non\_immediate\_instances \rightarrow \langle O \rangle]$:-
    $O :* C, \neg O : C$

$C[instances \rightarrow \langle O \rangle]$:- $O :* C$,

$C[highest\_paid\_employee \rightarrow O]$:-
    $O : C[salary \rightarrow S1]$
    $\neg O' : C[salary \rightarrow S2]$,
    $S1 < S2$

The first five rules are the ROL versions of the F-logic rules in Example 4.6. The next four collect all immediate subclasses, all nonimmediate subclasses, all immediate instances and all nonimmediate instances of of each class, respectively. The last rule finds the employees who have the highest salary in their department.

As discussed in Section 3, ROL's functor classes can be used as relations. However, the higher-order mechanism used in ROL only allows us to query the relation schema and it is impossible to query just the attributes in the relations. In addition, the non-immediate and immediate features are cumbersome.

5. UPDATES

Like a relational database, a deductive database also undergoes updates to absorb new information. In the past decade, various methods to incorporate update constructs into logic-based languages have been proposed [Abiteboul 1988; Abiteboul and Vianu 1991; Bonner and Kifer 1993; Bonner et al. 1993; Bonner and Kifer 1994; Bry 1990; Chen 1997; de Maindreville and Simon 1988; Kakas and Mancarella 1990; Kowalski 1992; Manchanda 1989; Montesi et al. 1997; Naqvi and Krishnamurthy 1988; Naish et al. 1987; Reiter 1995; Wichert and Freitag 1997]. Indeed, modeling updates in logic-based languages is still the subject of substantial current research.

The important issues related to updates from a deductive database point of view are:

(1) extensional and intensional database updates,

(2) nondeterministic updates,

(3) bulk or set-oriented updates,

(4) conditional updates.

In this section, we discuss four families of deductive languages that support updates, Prolog, DLP and its extension, DatalogA and LDL, and Transaction Logic, and show how they deal with the above issues.

5.1 Prolog

In Prolog, the basic update primitives are *assert* and *retract*. *Assert* is used to insert a single fact or rule into the database. It always succeeds initially and fails when the computation backtracks. Facts and rules can be deleted from the database in Prolog by using *retract*. Initially, *retract* deletes the first clause in the database that unifies with the argument of *retract*. On backtracking, the next matching clause is removed. It fails when there are no remaining matching clauses. Therefore, both extensional and intensional databases can be updated with Prolog.

Example 5.1 Let *employee* be a base relation that stores the department and salary of each employee. The following query in Prolog is used to fire employee

Joe in the Toy department with a salary of 10K and hire an employee Bob in the Shoe department with a salary of 15K.

$? - retract(employee(joe, toy, 10K)),$
$\quad assert(employee(bob, shoe, 15K))$

Updates expressed in queries can be performed only once. If the same kind of updates may be used more than once, it is better to write a general update rule and then query the head of the rule for specific updates.

Example 5.2 Let *account* be a base relation that stores the balance of each account. The following rule can be used to withdraw money from an account:

$withdraw(Acc, Amt)$:-
$\quad account(Acc, Bal_1),$
$\quad Bal_1 \geq Amt,$
$\quad Bal_2 = Bal_1 - Amt,$
$\quad retract(account(Acc, Bal_1)),$
$\quad assert(account(Acc, Bal_2))$

If we want to withdraw \$30 from the account $a123$, then we can use a query ?- $withdraw(a123,30)$.

In Prolog, variables that occur only in the body of a rule are existentially quantified. Therefore, updates in Prolog can be nondeterministic.

Example 5.3 Let *course*, *section* and *size* be base relations where *course* stores the sections for each course, *section* stores students for each section, and *size* stores the size of each section. The following Prolog rule can enroll a student into one section of the course whose size is no more than 30 nondeterministically:

$enroll(Student, Course)$:-
$\quad course(Course, Section),$
$\quad size(Section, Num_1),$
$\quad Num_1 < 30, Num_2 = Num_1 + 1,$
$\quad retract(size(Section, Num_1)),$
$\quad assert(section(Section, Student)),$
$\quad assert(size(Section, Num_2))$

The nondeterminism is due to the variable *Section*, which occurs only in the body of the rule and is therefore existentially quantified.

Prolog uses a top-down, tuple-at-a-time, backtracking-based inference mechanism for updates and queries. For bulk updates, it resorts to recursive update rules in Prolog.

Example 5.4 Let *employee* be a base relation as in Example 5.1. The following recursive rules can be used to give every employee an *X%* salary increase in Prolog:

$raise(X)$:- $\neg employee(Name, Dept, Sal)$

$raise(X)$:-
$\quad retract(employee(Name, Dept, Sal_1)),$
$\quad Sal_2 = Sal_1 + Sal_1{*}X,$
$\quad raise(X),$
$\quad assert(employee(Name, Dept, Sal_2))$

In order to give every employee a 10% salary increase, we can issue a query ?- $raise(0.1)$. If there are still employees in the database, the second will nondeterministically delete an employee, call the query ?- $raise(0.1)$ recursively, and then insert the employee with a new salary. In this way, the salary of each employee can be increased.

Unfortunately, the semantics of *assert* and *retract* is not well defined in Prolog. The exact effect of calling code containing *assert* and *retract* is often difficult to predict. The resulting database update relies only on the operational semantics of Prolog, rather than just the declarative semantics. The order of execution of subgoals is as important as the logical content of the goal. For the rules in Examples 5.2, 5.3 and 5.4, changing the order of emph assert or *retract* in the body of the rules may yield different results. Many distributed Prolog systems have versions of *retract* with bugs or strange behavior (sometimes called "features"). In an external or distributed database system, the problems with *assert* and *retract* become much more severe. Multiuser access creates even more difficulties. In addition, the tuple-at-a-time, backtracking-based update mechanism of Prolog is not suitable for deductive databases. The following example shows another problem with conditional updates in Prolog.

Example 5.5 Let *employee* be as in Example 5.1 and *avg_sal* an intensional relation based on *employee* that gives the average salary of employees in each department. Consider the following Prolog rule:

$$\begin{aligned}&hire(Name, Dept, Sal) \text{ :-}\\ &\quad assert(employee(Name, Dept, Sal)),\\ &\quad avg\_sal(Dept, Avg), Avg \leq 50K\end{aligned}$$

The intention here is to hire an employee only if the average salary in the department stays below 50K after hiring. However, in Prolog, if the average salary after hiring Bob is greater than 50K, Bob has still been hired, since Prolog does not undo updates during backtracking.

5.2 DLP and Its Extension

The notion of states is inherent in any notion of updates. A state in Datalog is a set of base and derived relations. Updates cause transitions from a state through a state space, thereby changing the database. Dynamic logic [Harel 1979] is a logic for reasoning about programs that test and change the values of an environment. It provides a natural state-transition semantics for updates and is therefore used in several extensions of Datalog with updates.

DLP (Dynamic Logic Programming) [Manchanda and Warren 1988] is such a Datalog extension. It allows only the extensional database to be updated. The intensional database cannot be updated.

In DLP, predicate symbols are divided into three disjoint sets: base predicate symbols, rule predicate symbols and dynamic predicate symbols. For every base predicate symbol p, two associated dynamic predicate symbols $+p$ and $-p$ are used to specify basic addition and deletion of tuples in the base relation p, respectively. Atoms formed using dynamic predicate symbols are called dynamic atoms.

An update query in DLP is of the form $\langle D \rangle(\alpha)$ where D is a dynamic atom used for updates and α is a query. The updates specified in D are executed only if the query α is evaluated true after the updates. If α is just *true*, then we can simply use $\langle D \rangle$ as an update query. As α can contain an update query, complex updates can be expressed.

The updates in Example 5.1 can be represented as a nested update query in DLP in two different ways as follows:

$$\begin{aligned}&? - \langle - employee(joe, toy, 10K)\rangle\\ &\quad (\langle + employee(bob, shoe, 15K)\rangle)\end{aligned}$$

$$\begin{aligned}&? - \langle - employee(bob, shoe, 15K)\rangle\\ &\quad (\langle + employee(joe, toy, 10K)\rangle)\end{aligned}$$

Update rules whose body contains update query can be used in DLP. The updates in Example 5.2 can be represented as an update rule in DLP as follows:

$$\begin{aligned}&\langle withdraw(Acc, Amt)\rangle \text{ :-}\\ &\quad \langle - account(Acc, Bal_1)\rangle\\ &\quad (Bal_1 \geq Amt, Bal_2 = Bal_1 - Amt,\\ &\quad \langle + account(Acc, Bal_1)\rangle)\end{aligned}$$

To withdraw \$30 from the account $a123$, we can use an update query ?- $\langle withdraw(a123,30)\rangle$.

As in Prolog, a top-down, tuple-at-a-time, backtracking-based inference mechanism is used in DLP for both updates and queries. However, unlike Prolog, it undoes updates during backtracking. For the above query ?$-\langle withdraw$ $(a123,30)\rangle$, DLP first deletes the tuple $account(a123, Bal_1)$. If $Bal_1 \geq 30$ is not true, DLP undoes the deletion during backtracking.

Like Prolog, DLP also supports nondeterministic updates as variables occurring in the body of a rule are existentially quantified. The nondeterministic update rule in Example 5.3 can be defined in DLP with a different ordering as follows:

$\langle enroll(Student, Course)\rangle$:-
  $course(Course, Section)$,
  $size(Section, Num_1)$,
  $\langle + section(Section, Student)\rangle$
  $(Num_1 < 30, Num_2 = Num_1 + 1$,
  $\langle - size(Section, Num_1)\rangle$
  $(\langle + size(Section, Num_2)\rangle))$

If a course DB has two sections, the query $? - enroll(joe, db)$ inserts Joe into one section and then checks if it is full. If so, it backtracks, undoes the insertion, and tries another section. Therefore, if neither section is full, Joe will be enrolled nondeterministically in a section.

As an update query in DLP contains a query part that must be evaluated true, we can express conditions that completed updates must satisfy. For example, the conditional updates in Example 5.4 can be represented as follows:

$\langle hire(Name, Dept, Sal)\rangle$:-
  $\langle + employee(Name, Dept, Sal)\rangle$
  $(avg\_sal(Dept, Ave), Avg \leq 50K$

Given a query $? - \langle hire(bob, toy, 15K)\rangle$, if the average salary after hiring Bob is greater than 50K, then $Avg \leq 50K$ will fail. Therefore, the whole query will fail and the update will be undone without leaving a residue in the database.

As in Prolog, the bulk updates in Example 5.5 must be represented in DLP by recursive update rules as follows:

$\langle raise(X)\rangle$:- $\neg employee(Name,Dept,Sal)$
$\langle raise(X)\rangle$:-
  $\langle - employee(Name, Dept, Sal_1)\rangle$
  $(Sal_2 = Sal_1 + Sal_1{*}X,)$
  $\langle raise(X)\rangle$
  $(\langle + employee(Name, Dept, Sal_2)\rangle)$

Updates in DLP have an operational semantics with a left-to-right reading. In addition, its tuple-at-a-time, backtracking-based update mechanism is not suitable for deductive databases. As a result, DLP is extended in Manchanda [1989] to directly support bulk updates. For example, the updates in Example 5.5 can be represented in this extension of DLP as follows:

$raise(X)$: $\{employee2(Name, Dept, Sal_2)$:-
  $employee(Name, Dept, Sal_1)$,
  $Sal_2 = Sal_1 + Sal_1{*}X\}$

Given a query ?- $raise(0.1)$, the updates are performed as follows: transform the given definition into the Datalog one:

$raise\_employee2(X, Name, Dept, Sal_2)$:-
  $employee(Name, Dept, Sal_1)$,
  $Sal_2 = Sal_1 + Sal_1{*}X$

Then evaluate the relation denoted by *raise_employee2* in the current database; project out the first attribute (corresponding to X) from this relation; replace the current instance of *employee* with this projection.

5.3 DatalogA and LDL

DatalogA (Datalog with Actions) [Naqvi and Krishnamurthy 1988] is another extension of Datalog with updates only to base relations. Like DLP, it uses dynamic logic to assign state-transition semantics to updates.

In DatalogA, base facts can be updated by using primitive update actions that are represented by prefixing a symbol + or − to a base predicate, where + is for insertion and − is for deletion. For example, $+ employee(joe, toy,$

$50K$) denotes the action that inserts the base fact $employee(joe, toy, 50K)$ into the current state.

Two kinds of updates are distinguished in DatalogA and have different semantics. The first kind is the update in which all different orders of execution yield the same final state. This kind of update has a declarative semantics and is represented by α, β, where α and β stand for update actions. The other kind is the update in which the update results depend on the order of execution. That is, different orders of execution may yield different final states. This kind of update has an operational semantics and is represented by $\alpha;\beta$, where α and β stand for update actions. The semantics for this kind does not require that α executed before or after β gives the same result.

The updates in Example 5.1 can be represented as a query in DatalogA as follows:

$$? - employee(joe, toy, 10K),$$
$$+ employee(bob, shoe, 15K)$$

Example 5.6 If we want to fire Joe and give his salary to Bob, we can use the following query:

$$? - employee(joe, toy, Sal);$$
$$+ employee(bob, shoe, Sal)$$

In this query we cannot change the order of actions, as we need to execute the deletion first to get the substitution for the variable Sal, and then perform the insertion.

Rules with update actions in the body can also be defined in DatalogA. For example, the update rule in Example 5.2 can be represented as follows:

```
withdraw(Acc, Amt) :-
      account(Acc, Bal_1),
      Bal_1 ≥ Amt,
      Bal_2 = Bal_1 − Amt,
      (− account(Acc, Bal_1);
      + account(Acc, Bal_2))
```

In DatalogA, rules with update actions are evaluated only if the head of the rule is queried (i.e., top-down), while rules without update actions are evaluated bottom-up even if they are not directly queried.

Unlike Prolog and DLP, in which variables occurring only in the body of an update rule are existentially quantified, such variables in DatalogA are universally quantified so that a set-oriented non-backtracking-based inference mechanism can be used for updates. For example, the updates in Example 5.4 can be represented as follows:

```
raise(X) :-
      employee(Name, Dept, Sal_1),
      Sal_2 = Sal_1 + Sal_1*X,
      ( − employee(Name, Dept, Sal_1);
       + employee(Name, Dept, Sal_2))
```

In this update rule, $employee(Name, Dept, Sal_1)$ is a query that generates all substitutions for $Name$, $Dept$, and Sal_1, and $(- employee(Name, Dept, Sal_1); + employee(Name, Dept, Sal_2))$, is used to change the salary of the employee from Sal_1 to Sal_2 for each group of substitutions.

Like Prolog, DatalogA does not support conditional updates, as it has no provision for backtracking and removing the effects of previous updates if the condition fails. Therefore, the intended updates in Example 5.5 cannot be expressed in DatalogA.

As the semantics for variables occurring only in the body and inference mechanism in DatalogA are different from those in Prolog, the update results can be quite different too, even though the rules look similar. The following rule is the DatalogA version of the rule in Example 5.3:

```
enroll(Student, Course) :-
      course(Course, Section),
      size(Section, Num_1),
      Num_1 < 30, Num_2 = Num_1 + 1,
      − size(Section, Num_1),
      + section(Section, Student),
      + size(Section, Num_2).
```

If a course DB has two sections, then the query ?- *enroll*(*joe*, *db*) will insert Joe into both sections of the DB course. Indeed, nondeterministic updates are not supported in DatalogA.

The update mechanism of DatalogA is incorporated in LDL. The nondeterministic updates in LDL are supported through the use of the *choice* operator of the form $choice((\overline{X}), (\overline{Y}))$, where $\overline{X}$ and $\overline{Y}$ are vectors of terms, which selects those instantiations of the variables $\overline{X}$ and $\overline{Y}$ that satisfy the functional dependency $\overline{X} \rightarrow \overline{Y}$. The nondeterministic updates in Example 5.3 can be represented in LDL as follows:

$$
\begin{aligned}
&enroll(Student, Course) \text{ :-}\\
&\quad course(Course, Section),\\
&\quad size(Section, Num_1),\\
&\quad Num_1 < 30, Num_2 = Num_1 + 1,\\
&\quad choice((Course), (Section)),\\
&\quad -\, size(Section, Num_1),\\
&\quad +\, section(Student, Section),\\
&\quad +\, size(Section, Num_2)
\end{aligned}
$$

The choice operator here guarantees that only one section of the course is selected for insertion.

The dynamic logic interpretation of updates in DatalogA and LDL provides a semantics to updates that is consonant with the operational meanings of the update predicates. However, this semantics is rather tricky.

5.4 Transaction Logic

The notion of transaction is essential for database operations. It was first introduced into Prolog informally in Naish et al. [1987]. Its use in logic programming was then formalized in Transaction Logic $T_{\mathcal{R}}$ [Bonner and Kifer 1993; 1994; Bonner et al. 1993]. $T_{\mathcal{R}}$ is an extension of first-order logic, both syntactically and semantically. Not only the extensional database but also the intensional database can be updated in $T_{\mathcal{R}}$. Unlike other approaches, the main novel feature of $T_{\mathcal{R}}$ is that it has a natural model theory and a sound and complete proof theory.

In $T_{\mathcal{R}}$, two primitive operations are provided to delete and insert facts and rules: *del* and *ins*. A new logical connective called *serial conjunction* and denoted by $\otimes$ is introduced to sequence transactions, where $\alpha \otimes \beta$ means do α then do β. The updates in Example 5.1 do not require sequencing and can be represented as a transaction in $T_{\mathcal{R}}$ as follows:

$$
\begin{aligned}
?-\ &del : employee(joe, toy, 10K),\\
&ins : employee(joe, shoe, 10K)
\end{aligned}
$$

The updates in Example 5.6 require sequencing and can be represented using $\otimes$ as follows:

$$
\begin{aligned}
?-\ &del : employee(joe, toy, Sal) \otimes\\
&ins : employee(joe, shoe, Sal)
\end{aligned}
$$

In $T_{\mathcal{R}}$, transactions can be defined using rules that can call other transactions. The update rule in Example 5.2 can be represented as follows:

$$
\begin{aligned}
&withdraw(Acc, Amt) \text{ :-}\\
&\quad balance(Acc, Bal) \otimes\\
&\quad Bal \geq Amt \otimes\\
&\quad del : balance(Acc, Bal) \otimes\\
&\quad ins : balance(Acc, Bal - Amt)
\end{aligned}
$$

As in Prolog and DLP, in $T_{\mathcal{R}}$ variables occurring only in the body of a rule are existentially quantified and tuple-at-a-time evaluation mechanism is used. However, the evaluation can be executed either bottom-up or top-down. Nondeterministic updates are naturally supported in this way. The rule in Example 5.3 can be represented in $T_{\mathcal{R}}$ as follows:

$$
\begin{aligned}
&enroll(Student, Course) \text{ :-}\\
&\quad (course(Course, Section),\\
&\quad size(Section, Num),\\
&\quad Num < 30 \otimes\\
&\quad ins : section(Section, Student) \otimes\\
&\quad del : size(Section, Num) \otimes\\
&\quad ins : size(Section, Num + 1).
\end{aligned}
$$

As a transaction must be done atomically, the conditional updates in Example 5.4 can be represented in $T_{\mathcal{R}}$ as follows:

$$\begin{aligned}&hire(Name, Dept, Sal) \text{ :-}\\&\quad ins : employee(Name, Dept, Sal) \otimes\\&\quad avg\_sal(Dept, Ave) \otimes\\&\quad Avg \leq 50K\end{aligned}$$

As in DLP and DatalogA, if the average salary after hiring an employee is greater than 50K, the whole query will fail without leaving a residue in the database.

In order to support bulk updates, $T_{\mathcal{R}}$ uses relational assignment as in relational databases, in the same spirit as the extension of DLP discussed in Section 5.2. For example, the following rules in $T_{\mathcal{R}}$ can be used to give every employee a 10% salary increase.

$$\begin{aligned}&employee2(Name, Dept, Sal{*}1.1) \text{ :-}\\&\quad employee(Name, Dept, Sal)\end{aligned}$$

$$raise \text{ :- } [employee := employee2]$$

Given a query $?- \; raise$, the new contents of the *employee* relation are computed by the first rule and are held in the relation *employee*2. Then the relational assignment $[employee := employee2]$ in the second rule assigns the new contents into the relation *employee*.

Unfortunately, the relational assignment cannot pass arguments. Therefore, we cannot define the rule in Example 5.4 that gives every employee an $X\%$ salary increase in $T_{\mathcal{R}}$.

6. CONCLUSION

Approaches to deductive databases are torn by two opposing forces. On one side there are the stringent real-world requirements of actual databases. The requirements include efficient processing as well as the ability to express complex and subtle real-world relationships. On the other side are the simple and clear semantics of logic programming and its inference capabilities. The need for expressiveness has forced deductive database languages away from their simple roots in logic programming and the relational data model.

In this paper, we have discussed the problems with the standard deductive database language Datalog (with negation) from four different aspects: complex values, object orientation, higher-orderness and updates. In each case, we have examined four typical solutions.

There are three important areas that we have not discussed here, since little has been published to date. One is that none of the deductive database languages are computationally complete so not all programming tasks can be performed in a single framework. The second is that none of the deductive database languages support unknown/null values, which are common in database applications. Indeed, it is not clear how to derive information with rules when there are null values in the database. Finally, most object-oriented deductive languages are only structurally object-oriented, since methods and encapsulation, two important features of object orientation, are not properly supported. In Bertino and Montesi [1992], encapsulation is addressed in an object-oriented deductive framework, but the semantics is not defined.

We expect that a few more expressive deductive database languages will be developed that incorporate various important features to solve the problems discussed here. In the mean time, various implementation techniques will be investigated. Powerful deductive database systems will eventually be delivered to the database market to extend database applications further.

ACKNOWLEDGMENTS

The author would like to thank Michael Franklin, associate editor of *ACM Computing Surveys*, and reviewers for their critical reading of the article. Their many valuable comments and suggestions substantially helped to improve the quality and accuracy of this survey. Thanks also to Katrina Avery at *ACM Computing Surveys*, Cory Butz, and Shilpesh Katragadda for their corrections. This work was partly supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

ABITEBOUL, S. 1988. Updates, a New Database Frontier. In *Proceedings of the International Conference on Data Base Theory* (Pruges, Belgium). Springer-Verlag, New York, 1–18.

ABITEBOUL, S. AND BEERI, C. 1995. The Power of Languages for the Manipulation of Complex Values. *VLDB J. 4*, 4, 727–794.

ABITEBOUL, S., FISCHER, P. C., AND SCHEK, H., Eds. 1987. *Proceedings of the International Workshop on Theory and Applications of Nested Relations and Complex Objects in Databases*. (Darmstadt, Germany) Springer-Verlag, New York.

ABITEBOUL, S. AND GRUMBACH, S. 1991. A rule-based language with functions and sets. *ACM Trans. Database Syst. 16*, 1 (Mar.), 1–30.

ABITEBOUL, S. AND HULL, R. 1987. IFO: a formal semantic database model. *ACM Trans. Database Syst. 12*, 4 (Dec. 1987), 525–565.

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley, Reading, MA.

ABITEBOUL, S. AND KANELLAKIS, P. 1989. Object Identity as a Query Language Primitive. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (SIGMOD '89, Portland, Oregon, June 1989), J. Clifford, J. Clifford, B. Lindsay, D. Maier, and J. Clifford, Eds. ACM Press, New York, NY, 159–173.

ABITEBOUL, S. AND VIANU, V. 1991. Datalog extensions for database queries and updates. *J. Comput. Syst. Sci. 43*, 1 (Aug. 1991), 62–124.

AÏT-KACI, H. 1993. An Introduction to Life. In *Proceedings of the Joint International Conference and Symposium on Logic Programming* (Vancouver, Canada). MIT Press, Cambridge, MA, 506–524.

AÏT-KACI, H AND NASR, R 1986. Login: A logic programming language with built-in inheritance. *J. Logic Program. 3*, 3 (Oct. 1986), 185–215.

ALBANO, A., CARDELLI, L., AND ORSINI, R. 1985. GALILEO: a strongly-typed, interactive conceptual language. *ACM Trans. Database Syst. 10*, 2 (June 1985), 230–260.

ALBANO, A., GHELLI, G., AND ORSINI, R. 1995. Fibonacci: A Programming Language for Object Databases. *VLDB J. 4*, 3, 403–444.

APT, K. R., BLAIR, H. A., AND WALKER, A. 1988. Towards a theory of declarative knowledge. In *Foundations of deductive databases and logic programming*, J. Minker, Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, 89–148.

BANCILHON, F., MAIER, F., SAGIV, Y., AND ULLMAN, J. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the ACM Symposium on Principles of Database Systems* (PODS '86, Cambridge, Massachusetts). ACM Press, New York, NY, 1–16.

BANCILHON, F. AND RAMAKRISHNAN, R. 1986. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the conference on Management of data* (SIGMOD '86, Washington, D.C., May 28-30, 1986), C. Zaniolo, Ed. ACM Press, New York, NY, 16–52.

BARBACK, M., LOBO, J., AND LU, J. 1992. Minimizing Indefinite Information in Disjunctive Deductive Databases. In *Proceedings of the International Conference on Data Base Theory* (Berlin, Germany). Springer-Verlag, New York, 246–260.

BEERI, C. 1989. Formal Models for Object-Oriented Databases. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases* (Kyoto, Japan), W. Kim, J. Nicolas, and S. Nishio, Eds. North-Holland Publishing Co., Amsterdam, The Netherlands, 405–430.

BEERI, C. 1990. A formal approach to object-oriented databases. *Data Knowl. Eng. 5*, 2, 353–382.

BEERI, C., NAQVI, S., SHMUELI, O., AND TSUR, S. 1991. Set constructors in a logic database language. *J. Logic Program. 10*, 3/4 (Apr./May 1991), 181–232.

BEERI, C. AND RAMAKRISHNAN, R. 1991. On the power of magic. *J. Logic Program. 10*, 3/4 (Apr./May 1991), 255–299.

BERTINO, E. AND MONTESI, D. 1992. Towards a Logical Object-oriented Programming Language for Databases. In *Proceedings of the International Conference on Extending Database Technology* (Vienna, Austria). Springer-Verlag, New York, NY, 168–183.

BONNER, A. J. 1989. Hypothetical datalog negation and linear recursion. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems* (PODS '89, Philadelphia, PA, Mar. 29–31, 1989), A. Silberschatz, Ed. ACM Press, New York, NY, 286–300.

BONNER, A. J. 1990. Hypothetical datalog: complexity and expressibility. *Theor. Comput. Sci. 76*, 1 (Oct. 31, 1990), 3–51.

BONNER, A. J. AND KIFER, M. 1993. Transaction logic programming. In *Proceedings of the tenth international conference on Logic programming* (ICLP'93, Budapest, Hungary, June 21–25, 1993), D. S. Warren, Ed. MIT Press, Cambridge, MA, 257–279.

BONNER, A. J. AND KIFER, M. 1994. An overview of transaction logic. *Theor. Comput. Sci. 133*, 2 (Oct. 24, 1994), 205–265.

BONNER, A., KIFER, M., AND CONSENS, M. 1993. Database Programming in Transaction Logic. In *Proceedings of the International Workshop on Database Programming Languages* (New York, NY). Morgan Kaufmann Publishers Inc., San Francisco, CA, 309–337.

BORGIDA, A. 1988. Modeling class hierarchies with contradictions. In *Proceedings of the Conference on Management of Data* (SIGMOD '88, Chicago, IL, June 1-3, 1988), H. Boral and P.-A. Larson, Eds. ACM Press, New York, NY, 434–443.

BRODIE, M. 1984. On the development of data models. In *On Conceptual Modelling*, M. Brodie, J. Mylopoulos, and J. Schmidt, Eds. Springer-Verlag, New York, NY, 19–48.

BRODSKY, A., JAFFAR, J., AND MAHER, M. 1993. Towards practical constraint databases. In *Proceedings of the International Conference on Very Large Data Bases* (Dublin, Ireland). Morgan Kaufmann Publishers Inc., San Francisco, CA, 557–580.

BRY, F. 1990. Intensional Updates: Abduction via Deduction. In *Proceedings of the International Conference on Logic Programming* (Budapest, Hungary). MIT Press, Cambridge, MA, 561–575.

CACACE, F., CERI, S., CREPI-REGHIZZI, S., TANCA, L., AND ZICARI, R. 1990. Integrating Object-Oriented Data Modelling with a Rule-Based Programming Paradigm. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (SIGMOD '90, Atlantic City, NJ, May 23–25, 1990), H. Garcia-Molina, Ed. ACM Press, New York, NY, 225–236.

CAREY, M. J., DEWITT, D. J., AND VANDERBERG, S. 1988. A data model and query language for EXODUS. In *Proceedings of the Conference on Management of Data* (SIGMOD '88, Chicago, IL, June 1-3, 1988), H. Boral and P.-A. Larson, Eds. ACM Press, New York, NY, 413–423.

CATTELL, R., Ed. 1996. *The Object Database Standard: ODMG-93*. Release 1.2 Morgan Kaufmann Publishers Inc., San Francisco, CA.

CERI, S., GOTTLOB, G., AND TANCA, L. 1990. *Logic programming and databases*. Springer-Verlag, New York, NY.

CHEN, P. 1976. The Entity-Relationship model: Toward a unified view of data. *ACM Trans. Database Syst. 1*, 1, 9–36.

CHEN, Q. AND CHU, W. 1989. HILOG: A High-Order Logic Programming Language for Non-1NF Deductive Databases. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases* (Kyoto, Japan), W. Kim, J. Nicolas, and S. Nishio, Eds. North-Holland Publishing Co., Amsterdam, The Netherlands, 431–452.

CHEN, Q. AND KAMBAYASHI, Y. 1991. Nested Relation Based Database Knowledge Representation. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (SIGMOD '91, Denver, CO, May 29–31, 1991), J. Clifford and R. King, Eds. ACM Press, New York, NY, 328–337.

CHEN, W. 1997. Programming with Logical Queries, Bulk Updates and Hypothetical Reasoning. *IEEE Trans. Knowl. Data Eng. 9*, 4, 587–599.

CHEN, W., KIFER, M., AND WARREN, D. S. 1993. HILOG: a foundation for higher-order logic programming. *J. Logic Program. 15*, 3 (Feb. 1993), 187–230.

CHEN, W. AND WARREN, D. S. 1989. C-logic of complex objects. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems* (PODS '89, Philadelphia, PA, Mar. 29–31, 1989), A. Silberschatz, Ed. ACM Press, New York, NY, 369–378.

CHIMENTI, D., GAMBOA, R., KRISHNAMURTHY, R., NAQVI, S., TSUR, S., AND ZANIOLO, C. 1990. The LDL System Prototype. *IEEE Trans. Knowl. Data Eng. 2*, 1, 76–90.

CODD, E. 1970. A relational model for large shared databases. *Commun. ACM 13*, 6, 377–387.

CODD, E. F. 1979. Extending the Database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 4 (Dec.), 397–434.

COLBY, L. 1989. A Recursive Algebra and Query Optimization for Nested Relations. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (SIGMOD '89, Portland, Oregon, June 1989), J. Clifford, J. Clifford, B. Lindsay, D. Maier, and J. Clifford, Eds. ACM Press, New York, NY, 124–138.

COLMERAUER, A. 1985. Prolog in 10 figures. *Commun. ACM 28*, 12 (Dec. 1985), 1296–1310.

DE MAINDREVILLE, C. AND SIMON, E. 1988. Modelling Non Deterministic Queries and Updates in Deductive Databases. In *Proc. 14th International Conference on Very Large Data Bases* (Los Angeles, CA). Morgan Kaufmann Publishers Inc., San Francisco, CA, 395–406.

DERR, M., MORISHITA, S., AND PHIPPS, G. 1993. Design and Implementation of the Glue-Nail Database System. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (SIGMOD '93, Washington, DC, May 26–28, 1993), P. Buneman and S. Jajodia, Eds. ACM Press, New York, NY, 147–167.

DEUX, O. 1991. The O2 system. *Commun. ACM 34*, 10 (Oct. 1991), 34–48.

DOVIER, A., OMODEO, E., PONTELLI, E., AND ROSSI, G. 1996. {log}: A Language for Programming in Logic with Finite Sets. *J. Logic Program. 28*, 1, 1–44.

FERNÁNDEZ, J. AND MINKER, J. 1992a. Semantics of Disjunctive Deductive Databases. In *Proceedings of the International Conference on Data Base Theory* (Berlin, Germany). Springer-Verlag, New York, 21–50.

FERNÁNDEZ, J. A. AND MINKER, J. 1992b. Disjunctive Deductive Databases. In *Proceedings of the International Conference on Logic Programming and Automated Reasoning* (LPAR '92, St. Peterburg, Russia). Springer-Verlag, New York, 332–356.

FISHMAN, D. H., B., B., CATE, H. P., CHOW, E. C., CONNORS, T., DAVIS, J. W., DERRETT, N., HOCH, C. G., KENT, W., LYNGBAEK, P., MAHBOD, B., NEIMAT, M. A., RYAN, T. A., AND SHAN, M. C. 1987. Iris: An object-Oriented Database Management System. *ACM Trans. Off. Inf. Syst. 5*, 1, 48–69.

FREITAG, B., SCHUTZ, H., AND SPECHT, G. 1991. LOLA—A Logic Language for Deductive Databases and its Implementation. In *Proceedings of the 2nd International Symposium on Database Systems for Advanced Applications* (DASFAA '91, Tokyo, Japan, Apr.). 216–225.

FROHN, J., HIMMERÖDER, R., KANDZIA, P., LAUSEN, G., AND SCHLEPPHORST, C. 1997. Florid: A Prototype for F-logic. In *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society, New York, NY.

GALLAIRE, H. 1981. Impacts of Logic on Data Bases. In *Proceedings of the International Conference on Very Large Data Bases* (Cannes, France). IEEE Computer Society, New York, NY, 248–259.

GALLAIRE, H. AND MINKER, J., Eds. 1978. *Logic and Data Bases*. Plenum Press, New York, NY.

GALLAIRE, H., MINKER, J., AND NICOLAS, J. M. 1984. Logic and Databases: A Deductive Approach. *ACM Comput. Surv. 16*, 2, 153–186.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming* (Washington, DC). MIT Press, Cambridge, MA, 1070–1080.

GRANT, J. AND MINKER, J. 1992. The impact of logic programming on databases. *Commun. ACM 35*, 3 (Mar. 1992), 66–81.

GRUMBACH, S. AND SU, J. 1996. Towards practical constraint databases (extended abstract). In *Proceedings of the fifteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems* (PODS '96, Montreal, P. Q., Canada, June 3–5, 1996), R. Hull, Ed. ACM Press, New York, NY, 28–39.

HAMMER, M. AND MCLEOD, D. 1981. Database description with SDM: A semantic database model. *ACM Trans. Database Syst. 6*, 3 (Sept.), 351-386.

HAN, J., LIU, L., AND XIE, Z. 1994. LogicBase: a deductive database system prototype. In *Proceedings of the 3rd International Conference on Information and Knowledge Management* (CIKM '94, Gaithersburg, Maryland, Nov. 29–Dec. 2, 1994), N. R. Adam, B. K. Bhargava, and Y. Yesha, Eds. ACM Press, New York, NY, 226–233.

HAREL, D. 1979. *First-Order Dynamic Logic*. Lecture Notes in Computer Science, vol. 361. Springer-Verlag, New York.

HEUER, A. AND SANDER, P. 1993. The LIVING IN A LATTICE Rule Language. *Data Knowl. Eng. 9*, 4, 249–286.

HILL, P. AND LLOYD, J. 1994. *The Gödel programming language*. MIT Press, Cambridge, MA.

INOUE, K. 1994. Hypothetical Reasoning in Logic Programs. *J. Logic Program. 18*, 3, 197–227.

IOANNIDIS, Y. AND RAMAKRISHNAN, R. 1988. Efficient Transitive Closure Algorithms. In *Proc. 14th International Conference on Very Large Data Bases* (Los Angeles, CA). Morgan Kaufmann Publishers Inc., San Francisco, CA, 382–394.

ISHIKAWA, H., SUZUKI, F., KOZAKURA, F., MAKINOUCHI, A., MIYAGISHIMA, M., IZUMIDA, Y., AOSHIMA, M., AND YAMANE, Y. 1993. The model, language, and implementation of an object-oriented multimedia knowledge base management system. *ACM Trans. Database Syst. 18*, 1 (Mar. 1993), 1–50.

JACOBS, B. 1982. On Database Logic. *J. ACM 29*, 2 (Apr.), 310–332.

JAFFAR, J. AND MAHER, M. 1994. Constraint logic programming: A survey. *J. Logic Program. 19-20*, 503-581.

JARKE, M., GALLERSDÖRFER, R., JEUSFELD, M. A., STAUDT, M., AND EHERER, S. 1995. ConceptBase—a deductive object base for meta data management. *J. Intell. Inf. Syst. 4*, 2 (Mar. 1995), 167–192.

JAYARAMAN, B. 1992. Implementation of subset-equational programs. *J. Logic Program. 12*, 4 (Apr. 1992), 299–324.

JIANG, B. 1990. A Suitable Algorithm for Computing Partial Transitive Closures. In *Proc. IEEE Sixth International Conference on Data Engineering* (Kobe, Japan, May). IEEE Computer Society, New York, NY, 264–271.

KAKAS, A. AND MANCARELLA, P. 1990. Database Updates through Abduction. In *Proceedings of the 16th VLDB Conference on Very Large Data Bases* (VLDB, Brisbane, Australia). VLDB Endowment, Berkeley, CA, 650–661.

KIEßLING, W., SCHMIDT, H., STRAUB, W., AND DUNZINGER, G. 1994. DECLARE and SDS: Early Efforts to Commercialize Deductive Database Technology. *VLDB J. 3*, 2, 211–243.

KIFER, M. AND LAUSEN, G. 1989. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Schema. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data* (SIGMOD '89, Portland, Oregon, June 1989), J. Clifford, J. Clifford, B. Lindsay, D. Maier, and J. Clifford, Eds. ACM Press, New York, NY, 134–146.

KIFER, M., LAUSEN, G., AND WU, J. 1995. Logical foundations of object-oriented and frame-based languages. *J. ACM 42*, 4 (July 1995), 741–843.

KIFER, M. AND WU, J. 1993. A logic for programming with complex objects. *J. Comput. Syst. Sci. 47*, 1 (Aug. 1993), 77–120.

KIM, W. 1990a. *Introduction to Object-Oriented Databases*. MIT Press, Cambridge, MA.

KIM, W. 1990b. Object-Oriented Databases: Definition and Research Direction. *IEEE Trans. Knowl. Data Eng. 2*, 3 (Sept.), 327–341.

KOWALSKI, R. A. 1988. The early years of logic programming. *Commun. ACM 31*, 1 (Jan. 1988), 38–43.

KOWALSKI, R. 1992. Database updates in the event calculus. *J. Logic Program. 12*, 1/2 (Jan. 1992), 121–146.

KRISHNAMURTHY, R. AND NAQVI, S. 1988. Towards a Real Horn Clause Language. In *Proc. 14th International Conference on Very Large Data Bases* (Los Angeles, CA). Morgan Kaufmann Publishers Inc., San Francisco, CA, 252–263.

KUPER, G. M. 1990. Logic programming with sets. *J. Comput. Syst. Sci. 41*, 1 (Aug. 1990), 44–64.

LECLUSE, C. AND RICHARD, P. 1989. The O_2 Database Programming Language. In *Proceedings of the fifteenth international conference on Very large data bases* (Amsterdam, The Netherlands, Aug 22–25 1989), R. P. van de Riet, Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, 411–422.

LEVENE, M. AND LOIZOU, G. 1993. Semantics for null extended nested relations. *ACM Trans. Database Syst. 18*, 3 (Sept. 1993), 414–459.

LIU, M. 1995. Relationlog: A Typed Extension to Datalog with Sets and Tuples (Extended Abstract). In *Proceedings of the International Symposium on Logic Programming* (ILPS '95, Portland, Oregon, Dec. 4–7). MIT Press, Cambridge, MA, 83–97.

LIU, M. 1996. ROL: A Deductive Object Base Language. *Inf. Syst. 21*, 5, 431–457.

LIU, M. 1998a. An Overview of Rule-based Object Language. *J. Intell. Inf. Syst. 10*, 1, 5–29.

LIU, M. 1998b. Relationlog: A Typed Extension to Datalog with Sets and Tuples. *J. Logic Program. 36*, 3, 271–299.

LIU, M. AND SHAN, R. 1998. The Design and Implementation of the Relationlog Deductive Database System. In *Proceedings of the 9th International Workshop on Database and Expert System Applications* (DEXA Workshop '98, Vienna, Austria, Aug. 24–28). IEEE Computer Society Press, Los Alamitos, CA, 856–863.

LIU, M., YU, W., GUO, M., AND SHAN, R. 1998. ROL: A Prototype for Deductive and Object-Oriented Databases. In *Proceedings of the International Symposium on Database Engineering and Applications* (ICDE '98, Orlando, FL, Feb. 23–27). IEEE Computer Society Press, Los Alamitos, CA, 598.

LLOYD, J. W. 1987. *Foundations of Logic Programming*. 2nd ed. Springer-Verlag Symbolic Computation and Artificial Intelligence Series. Springer-Verlag, Vienna, Austria.

LOU, Y. AND OZSOYOGLU, M. 1991. LLO: A Deductive Language with Methods and Method Inheritance. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (SIGMOD '91, Denver, CO, May 29–31, 1991), J. Clifford and R. King, Eds. ACM Press, New York, NY, 198–207.

MAIER, D. 1983. *The Theory of Relational Databases*. Computer Science Press, Inc., New York, NY.

MAIER, D. 1986. A logic for objects. Technical Report CS/E-86-012. Oregon Graduate Center, Beaverton, Oregon.

MAIER, D. 1987. Why Database Languages are a Bad Idea. In *Proceedings of the Workshop on Database Programming Languages* (Roscoff, France).

MAIER, D., STEIN, J., OTIS, A., AND PURDY, A. 1986. Development of Object-Oriented DBMS. In *OOPSLA '86*. ACM Press, New York, NY, 472–482.

MANCHANDA, S. 1989. Declarative expression of deductive database updates. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems* (PODS '89, Philadelphia, PA, Mar. 29–31, 1989), A. Silberschatz, Ed. ACM Press, New York, NY, 93–100.

MANCHANDA, S. AND WARREN, D. S. 1988. A logic-based language for database updates. In *Foundations of deductive databases and logic programming*, J. Minker, Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, 363–394.

MONTESI, D., BERTINO, E., AND MARTELLI, M. 1997. Transactions and Updates in Deductive Databases. *IEEE Trans. Knowl. Data Eng. 9*, 5, 784–797.

MORRIS, K., ULLMAN, J. D, AND VAN GELDER, A. 1986. Design overview of the NAIL! system. In *Proceedings on Third international conference on logic programming* (London, UK, July 14-18, 1986), E. Shapiro, Ed. Springer-Verlag, New York, NY, 554–568.

MOSS, C. 1994. *Prolog++*. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

MUMICK, I. S., FINKELSTEIN, S. J., PIRAHESH, H., AND RAMAKRISHNAN, R. 1996. Magic conditions. *ACM Trans. Database Syst. 21*, 1, 107–155.

Mylopoulos, J., Bernstein, P. A., and Wong, H. K. T. 1980. A Language Facility for Designing Database-Intensive Applications. *ACM Trans. Database Syst. 5*, 2 (June), 185–207.

Naish, L., Thom, L., and Ramamohanarao, K. 1987. Concurrent Database Updates in Prolog. In *Proceedings of the International Conference on Logic Programming* (Melbourne, Australia). MIT Press, Cambridge, MA, 178–189.

Naqvi, S. and Krishnamurthy, R. 1988. Database Updates in Logic Programming. In *Proc. 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Austin, TX, Mar.). 261–272.

Naqvi, S. and Tsur, S. 1989. *A logical language for data and knowledge bases*. Computer Science Press principles of computer science series. Computer Science Press, Inc., New York, NY.

Ozsoyoglu, Z. M. and Yuan, L.-Y. 1987. A new normal form for nested relations. *ACM Trans. Database Syst. 12*, 1 (Mar. 1987), 111–136.

Przymusinski, T. C. 1988. On the declarative semantics of deductive databases and logic programs. In *Foundations of deductive databases and logic programming*, J. Minker, Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, 193–216.

Przymusinski, T. 1990. Extended Stable Semantics for Normal and Disjunctive Programs.
In *Proceedings of the International Conference on Logic Programming* (Budapest, Hungary). MIT Press, Cambridge, MA, 459–477.

Ramakrishman, R., Ed. 1994. *Applications of Logic Databases*. Kluwer Academic Publishers, Hingham, MA.

Ramakrishnan, R., Srivastava, D., Sudarshan, S., and Seshadri, P. 1994. The coral deductive system. *VLDB J. 3*, 2, 161–210.

Ramakrishman, R. and Ullman, J. D. 1995. A Survey of Deductive Database Systems. *J. Logic Program. 23*, 2, 125–150.

Rao, P., Sagonas, K., Swift, T., Warren, D., and Freire, J. 1997. XSB: A System for Efficiently Computing. In *Proceedings of the International Conference on Logic Programming and Non-Monotonic Reasoning* (LPNMR '97, Dagstuhl, Germany). Springer-Verlag, New York, 431–441.

Reiter, R. 1984. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling*, M. Brodie, J. Mylopoulos, and J. Schmidt, Eds. Springer-Verlag, New York, NY, 191–233.

Reiter, R. 1995. On Specifying Database Updates. *J. Logic Program. 25*, 1, 53–91.

Ross, K. A. 1994. Modular stratification and magic sets for Datalog programs with negation. *J. ACM 41*, 6 (Nov. 1994), 1216–1266.

Roth, M. A., Korth, H. F., and Batory, D. S. 1987. SQL/NF: a query language for $\neg$ 1NF relational databases. *Inf. Syst. 12*, 1 (Jan. 1987), 99–114.

Roth, M. A., Korth, H. F., and Silberschatz, A. 1988. Extended algebra and calculus for $\neg$1NF relational databases. *ACM Trans. Database Syst. 13*, 4 (Dec. 1988), 389–417.

Sacca, D. and Zaniolo, C. 1987. Magic Counting Methods. In *Proceedings of the ACM SIGMOD Annual Conference on Management of Data* (SIGMOD '87, San Francisco, CA, May 27-29, 1987), U. Dayal, Ed. ACM Press, New York, NY, 49–59.

Sagonas, K., Swift, T., and Warren, D. 1994. XSB as an Efficient Deductive Database Engine. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (SIGMOD '94, Minneapolis, Minnesota, May 24–27, 1994), R. T. Snodgrass and M. Winslett, Eds. ACM Press, New York, NY, 442–453.

Sattar, A. and Goebel, R. 1997. Consistency-motivated reason maintenance in hypothetical reasoning. *New Gen. Comput. 15*, 2, 163–186.

Shan, R. and Liu, M. 1998. Introduction to the Relationlog System. In *Proceedings of the 6th Intl. Workshop on Deductive Databases and Logic Programming* (DDLP '98, Manchester, UK, June 20 1998). 71–83.

Shipman, D. W. 1981. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. Database Syst. 6*, 1, 140–173.

Smolka, G. 1995. The Oz programming model. In *Computer Science Today*, J. van Leeuwen, Ed. Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, New York, 324–343.

Somogyi, Z., Henderson, F., and Conway, T. 1996. The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language. *J. Logic Program. 29*, 1–3, 17–64.

Sterling, L. and Shapiro, E. 1986. *The art of Prolog: advanced programming techniques*. MIT Press series in logic programming. MIT Press, Cambridge, MA.

Su, S. Y. W. 1986. Modeling Integrated Manufacturing Data with SAM $*$. *IEEE Computer 19*, 1, 34–49.

Subrahmanian, V. S. 1992. Paraconsistent disjunctive deductive databases. *Theor. Comput. Sci. 93*, 1 (Feb. 3, 1992), 115–141.

Tsur, S. and Zaniolo, C. 1986. LDL: A Logic-based data-language. In *Proceedings of the 12th international conference on on Very Large Data Bases* (Kyoto, Japan, Aug., 1986). VLDB Endowment, Berkeley, CA, 33-41.

ULLMAN, J. 1982. *Principles of Database Systems*. Computer Science Press, Inc., New York, NY.

ULLMAN, J. 1989a. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Inc., New York, NY.

ULLMAN, J. D. 1989b. Bottom-up beats top-down for datalog. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems* (PODS '89, Philadelphia, PA, Mar. 29–31, 1989), A. Silberschatz, Ed. ACM Press, New York, NY, 140–149.

ULLMAN, J. 1991. A Comparison between Deductive and Object-Oriented Databases Systems. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases* (Munich, Germany), C. Delobel, M. Kifer, and Y. Masunaga, Eds. Springer-Verlag, New York, 263–277.

VAGHANI, J., RAMANOHANARAO, K., KEMP, D., SOMOGYI, Z., STUCKEY, P., LEASK, T., AND HARLAND, J. 1994. The Aditi Deductive Database System. *VLDB J. 3*, 2, 245–288.

VAN GELDER, A. 1993. The alternating fixpoint of logic programs with negation. *J. Comput. Syst. Sci. 47*, 1 (Aug. 1993), 185–221.

VAN GELDER, A., ROSS, K. A., AND SCHLIPF, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM 38*, 3 (July 1991), 619–649.

WICHERT, C.-A. AND FREITAG, B. 1997. Capturing Database Dynamics by Deferred Updates. In *Proceedings of the International Conference on Logic Programming* (Leuven, Belgium). MIT Press, Cambridge, MA.

Received: March 1997; revised: September 1997, April 1998; accepted: July 1998