Deductive Databases — Where to Now?

Mengchi Liu

Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2

Email: mliu@cs.uregina.ca
Phone: (306) 585-4700
Fax: (306) 585-4745

Submitted to JIIS

July 18, 1996

Abstract

Deductive databases result from the integration of database and logic programming techniques. It combines the benefits of these two approaches, such as representational and operational uniformity, recursion, declarative querying, efficient secondary storage access, etc. However, significant problems remain inherent in this synthesis. There are four broad areas where problems are apparent. The first problem area is how to naturally represent complex values consists of nested tuples and sets. The second problem area is how to incorporate object-oriented features into deductive framework. The third problem area is how to naturally deal with higher-order features in deductive databases. The last is the area of updates. This paper elaborates these problems, discusses some of the typical attempts that have been made towards solving them, and provides some perspective.

Keywords: Deductive databases, nested relational databases, complex object databases, object-oriented databases, logic programming.

1 Introduction

Databases and logic programming are two independently developed areas in computer science. Database technology has evolved in order to efficiently organize, manage and maintain large amounts of data. The relatively slow speed of secondary devices holding the data was one of the main limitations of database systems in the past. Hence, the internal organization of databases was the primary focus of research. Besides, the need to share information among a variety of users required strict rules governing the manipulation of data to be imposed to preserve the integrity of the database and to guarantee privacy for each user. This led to the development of several basic models. The most significant and widely used data model is the relational data model, which is based on mathematical relation theory. In the last two decades, a great deal of thought and ingenuity has been invested in the efficient processing of queries and updates over a wide range of storage structure and databases in secondary memory. A number of relational database systems such as Oracle, Sybase, Ingres have been widely used and more expressive data models and systems have been developed [4, 12, 13, 15, 17, 22, 23, 25, 27, 29, 35, 50, 56].

Logic programming began in the early seventies as a direct outgrowth of earlier work in automatic theorem proving and artificial intelligence. Logic programming is based on mathematical logic, which is formalized in terms of proof theory and model theory. 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. Most logic programming is based on the Horn-clause form, which is a subset of first-order logic. Horn-clause logic has well defined model-theoretic and procedural semantics [43]. The state of the art in logic programming is represented by Prolog in its various manifestations.

Important studies on the relationships between logic programming and relational databases have been conducted since the end of the seventies, mostly from a theoretical point of view. Relational database and logic programming techniques have been found to be strongly similar in their representation of data. They have also been found to be complementary. Relational database systems are superior to standard implementations of Prolog with respect to data independence, suitability for parallel processing and secondary storage access. The control over the execution of query languages is the responsibility of the system, which via query optimization and compilation techniques, ensures efficient performance over a wide range of storage structures and database demographies. However, the expressive power and functionality offered by relational database query languages is limited compared with the logic programming languages. Besides, relational query languages are often powerless to express complete applications, and are thus embedded in traditional programming languages. This method causes a so-called *impedance mismatch* [45] 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, and integrity constraints in a uniform way [53].

The combination of logic programming and relational database techniques has led to the active research area of deductive databases [24]. It combines the benefits of these two approaches, such as representational and operational uniformity, recursion, declarative querying, efficient secondary storage access, etc. The functors of Prolog, which are typically used for building recursive functions and complex data structures have been found not useful for operating over relational databases made up of flat relations. As a result, a restricted form of Prolog without functors 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 [16]. In recent years, various set-oriented evaluation strategies [6, 5, 16, 21, 28, 30, 58] specific for deductive databases have been the main focus of extensive research and a number of deductive database systems or prototypes based on Datalog, such as Nail[47], Aditi[59], Glue-Nail[21], XSB[52], LogicBase [26] etc. have been developed and reported. The 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, path problems [51]. They are best suited for applications in which a large amount of data must be accessed and complex queries must be supported.

Unfortunately, significant problems remain inherent in such kind of deductive databases. There are four broad areas where problems are apparent. The first problem area is that such deductive databases cannot meet the ever-increasing requirements of complex database applications as Datalog and the underlying relational data model use inexpressive flat structures, which cannot naturally model complex values. The second problem area is how to incorporate object-oriented features into deductive databases. The third problem area is that current deductive databases cannot naturally deal with schema and higher-order features. The fourth problem area is updates. It is necessary to be able to update and modify a database, but there is no immediately obvious way of expressing this within deductive framework.

In this paper, we elaborate these problems, discuss some of the typical attempts that have been made towards solving them, and provide some perspectives.

2 Complex-Value Deductive Languages

It has been realizing that Datalog and its underling relational data model using flat relations cannot provide natural and direct support for many novel advanced 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. Instead, Prolog can do better in this regard by supporting functors. For example, we may use the following facts in Prolog to store information about cars and their owners:

$$
\begin{array}{l}
car(vehicle(ny\text{-}858778,\ chevy\text{-}vega,\ 1983),\\
\quad owner(name(arthur, keller),\\
\qquad getFrom(dealer)))
\end{array}
$$

$$
\begin{array}{l}
car(vehicle(ny\text{-}358735,\ wolkswagen,\ 1990),\\
\quad owner(name(arthur, keller),\\
\qquad getFrom(owner(name(kurt, godel),
\end{array}
$$

$$getFrom(owner(name(norbert, wiener), \\ getFrom(dealer)))))))$$

where *vehicle, name, owner, getFrom* are functors which are not interpreted, i.e., they do not like the functors $+$, $-$, $*$, $\div$ which have specific interpretations. The above examples show that nested tuples can be supported indirectly by using functors in Prolog.

However, Prolog provides poor support for sets. In Prolog, the set expressions are represented using the built-in predicate *setof*:

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

which is read as "The set of instances of X such that P is provable is S". The term P represents a goal or goals. The term X is a variable occurring in P. The set S is represented as a list whose elements are sorted into a standard order without any duplicates.

However, the *setof* predicate in Prolog only has operational semantics. 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 sets. When a predicate involves more than one set, the program can become quite complicated and nonintuitive, which is contrary to the general philosophy of declarative programming [38].

LDL For the reasons discussed above, a powerful deductive database langauge called Logical Data Language (LDL) has been designed and implemented [8, 18, 57]. Like Prolog, LDL supports functors. Furthermore, it supports sets and provides two powerful set representation mechanisms: set enumeration and set grouping. Set enumeration is the process of constructing a set by listing its elements.

Example 2.1 If we want to derive a relation on sets of book titles from the *book* base relation such that their total price does not exceed \$100, we can use the following rule with set enumeration in LDL:

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

The same thing can not be directly represented in Prolog.

Unlike set-enumeration, in set-grouping of LDL, the set is constructed by defining its elements with a property that they must satisfy. Next example shows how to use grouping in LDL.

Example 2.2 Given the relation *parentsof* 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 special notation for set grouping:

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

However, there are three problems in LDL. First, the grouping mechanism is restricted to single rule rather than the program (a set of rules).

Example 2.3 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 $parents(bob, \{pam, tom\})$:

$$parentsof(X, \langle Y \rangle) :\!-\ fatherof(X, Y)$$
$$parentsof(X, \langle Y \rangle) :\!-\ motherof(X, Y)$$

as they actually generate $parents(bob, \{pam\})$ and $parents(bob, \{tom\})$ instead. We have to introduce an intermediate relation *parentof* and use it to obtain what we want as shown in Example 2.2.

The second problem is that the grouping requires stratfication.

Example 2.4 Consider the following rule:

$$ancestorsof(X, \langle Y \rangle) :\!-\ parentsof(X, \langle Z \rangle), ancestorsof(Z, S), Y \in S$$

Here we intend to define *ancestorsof* relations recursively. But the grouping mechanism of LDL requires we know the *parentsof* and *ancestorsof* relations in the body before we can compute the *ancestorsof* relation in the head which means recursive definition of the relation *ancestorsof* is not possible.

In [2, 19, 55], extended relational algebra operations such as *nest, unnest, union, intersect, difference, join* etc. are defined which extend the traditional relational algebra operations to nested relations. The third problem with LDL is that it is burdensome and ineffective to deal with these operations in LDL.

Example 2.5 Consider the following two relations:

p

a_1	b_1	c_1	d_1
a_1	b_1	c_1	d_2
a_1	b_2	c_1	d_1
a_1	b_2	c_1	d_2
a_1	b_2	c_2	d_3
a_2	b_1	c_1	d_1
a_2	b_1	c_1	d_3
a_2	b_3	c_2	d_1

$q = nest(p)$

<table>
<tr><td rowspan="5">a_1</td><td rowspan="2">b_1</td><td rowspan="2">c_1</td><td>d_1</td></tr>
<tr><td>d_2</td></tr>
<tr><td rowspan="3">b_2</td><td rowspan="2">c_1</td><td>d_1</td></tr>
<tr><td>d_2</td></tr>
<tr><td>c_2</td><td>d_3</td></tr>
<tr><td rowspan="3">a_2</td><td rowspan="3">b_1</td><td rowspan="2">c_1</td><td>d_1</td></tr>
<tr><td>d_3</td></tr>
<tr><td>c_2</td><td>d_1</td></tr>
</table>

Here p is a flat relation while q is a nested relation and q is obtained by applying the *nest* operation on p.

In order to perform the nest operation in LDL, we have to use grouping several times and introduce several intermediate functors and relations as follows:

$$
\begin{array}{l}
r1(A,B,C,\langle D\rangle) \text{ :- } p(A,B,C,D) \\
r2(A,B,f(C,E)) \text{ :- } r1(A,B,C,E) \\
r3(A,B,\langle F\rangle) \text{ :- } r2(A,B,F) \\
r4(A,g(B,G)) \text{ :- } r3(A,B,G) \\
q(A,\langle H\rangle) \text{ :- } p4(A,H)
\end{array}
$$

COL COL[3] is also an extension of Datalog intended to support complex values as well. Unlike LDL which use functor objects for tuples indirectly, COL directly supports tuples and tuple constructors. It supports sets as well. Furthermore. It interprets functors as mappings to sets and use them for the set grouping and enumeration purpose.

Consider the facts in Example 2.3 again. The *parentsof* relation can be defined in COL as follows:

$$
\begin{array}{l}
Y \in par(X) \text{ :- } fatherof(X,Y) \\
Y \in par(X) \text{ :- } motherof(X,Y) \\
parentsof(X, par(X))
\end{array}
$$

Here we use the functor *par* to group all parents of a person X into a set denoted by $par(X)$ and the grouping can be done using several rules. Furthermore, we can group a set recursively. The task shown in Example 2.4 can be done in COL as follows:

$$
\begin{array}{l}
Y \in anc(X) \text{ :- } Z \in par(X), Y \in anc(Z) \\
ancestors(X, anc(X))
\end{array}
$$

The set enumeration of LDL shown in Example 2.1 can be done in COL using a 0-ary functor f as follows:

$$
\begin{array}{l}
f \ni \{X,Y,Z\} \text{ :- } book(X,Px), book(X,Py), book(X,Pz), \\
\qquad\qquad\qquad\quad X \neq Y, X \neq Z, Y \neq Z, Px + Py + Pz < 100 \\
book\_deal(X) \in X \in f
\end{array}
$$

As a result, COL don't have the first and second problems of LDL. However, COL still has the third problem of LDL. Following rules in COL show how to nest the relation p to obtain the relation q shown in Example 2.5:

$$
\begin{array}{l}
D \in f(A,B,C) \text{ :- } p(A,B,C,D) \\
r1(A,B,C,f(A,B,C)) \\
[C,E] \in g(A,B) \text{ :- } r1(A,B,C,E) \\
r2(A,B,g(A,B)) \\
[B,F] \in h(A) \text{ :- } r2(A,B,F)
\end{array}
$$

$q(A, h(A))$

Therefore, using interpreted functors in COL to manipulate complex values is still cumbersome from user's point of view and ineffective from implementation point of view as intermediate mappings need to be generated.

Relationlog Relationlog [40] extends Datalog directly with powerful set and tuple constructors. It solves all three problems of LDL by generalizing LDL set grouping and set enumeration mechanisms into partial and complete set terms. More accurately, complete set terms are the same as LDL set enumeration expressions while partial set terms generalize the LDL set grouping expressions. Unlike set grouping expressions of LDL, partial set terms in Relationlog can appear not only in the head but also in the body of a rule. When in the body, it denotes part of a set. When in the head, it is used to group a set and several rules can be used to group the same set.

Consider Example 2.3 again: The *parentsof* relation can be directly defined in Relationlog as follows:

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

Here two rules are used to group the same set.

The use of partial set terms in the head doesn't require stratification. The *ancestorsof* relation in Example 2.4 can be defined recursively in Relationlog as follows:

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

Finally, the extended relational algebra operations can be represented in Relationlog in a simple, intuitive and direct way. The *nest* operation shown in Example 2.5 can be represented in Relationlog as follows:

$$q(A, \langle [B, \langle [C, \langle D \rangle] \rangle] \rangle) :- p(A, B, C, D)$$

3 Object-Oriented Deductive Languages

In this section, we discuss the problems with complex-value deductive languages, then introduce fundemantal principles of object-oriented paradigm and finally examine their applications in deductive frameworks.

With complex-value deductive languages, two facts cannot share subparts. Consider the following example.

Example 3.1 Suppose we have two facts in Prolog/LDL:

$$book(\text{“}Prolog\text{”}, author(name(\text{“}Bob\text{”}, \text{“}Su\text{”}), address(\text{“}257\ 9th\ Av\ SW\text{”})).$$
$$book(\text{“}Databases\text{”}, author(name(\text{“}Bob\text{”}, \text{“}Su\text{”}), address(\text{“}257\ 9th\ Av\ SW\text{”})).$$

both talking about the same individual, and if he moves, both facts should be updated. Further more, the change is made by retracting those facts and asserting new facts:

$book(\text{"}Prolog\text{"}, author(name(\text{"}Bob\text{"}, \text{"}Su\text{"}), address(\text{"}128\ 2nd\ St\ NW\text{"}))$.
$book(\text{"}Databases\text{"}, author(name(\text{"}Bob\text{"}, \text{"}Su\text{"}), address(\text{"}128\ 2nd\ St\ NW\text{"}))$.

In interpreting these new facts, nothing about them requires that the domain element representing $name(\text{"}Bob\text{"}, \text{"}Su\text{"})$ in the first case is the same as in the second. There is no way to say that everything stayed the same except the address. The problem here is that two facts cannot share subparts with Horn clause-based logic programming. The subpart is an object. We need something to identify it and we should be able to refer to its identity.

A solution to this problem requires explicitly using a non-logical identifier to represent each object. For the facts given in Example 3.1, we can use:

$book(\text{"}Prolog\text{"}, \text{"}m1\text{"})$.
$book(\text{"}Databases\text{"}, \text{"}m1\text{"})$.
$author(\text{"}m1\text{"}, name(\text{"}Bob\text{"}, \text{"}Su\text{"}), address(\text{"}257\ 9th\ Av\ SW\text{"}))$.

This is equivalent to relational normalization, i.e., split the original relation into two relations. This solution requires systematically introducing "meaningless" terms or keys such as *"m1"* to identify the corresponding object by the user. The problem with this approach is that the terms or keys are user specified and subject to updates. There ought to be some way of doing this without exposing the inner organization of the database. This capability is called object identity [32]. Object identity is the property of an object that distinguishes it from all others regardless of their content, location, or addressability. Object identity can only be created or destroyed but should not be subject to updates. Two criteria are used to measure the degree of content and location independence which the object identity provides: data independence which means that identity is preserved through changes in either data values or structure; location independence which means that identity is preserved through movement of objects among physical locations or address spaces. The solution given above which uses $m1$ to identify the object is not data independent as it is no different from other values and subject to updates. Any change to it will cause a discontinuity in identity.

With object identity, a natural question is what is an object? In O_2 [39], an object is pair of object identifier and value. The object is referenced with its object identifier which cannot be changed. The value associated with the object can be an integer, a string, a tuple, or a set, and is subject to change. The value can contain object identifiers so that two objects can share objects in this way and changes to the values of the shared objects won't affect the objects that refer to them. The value part of an object is called its state in [7].

Example 3.2 The facts given in Example 3.1 can be represented with object identifiers in O_2 as follows:

$prolog : [name : \text{"}Prolog\text{"}, author : bsu]$
$db : [name : \text{"}Databases\text{"}, author : bsu]$
$bsu : [name : [first : \text{"}Bob\text{"}, last : \text{"}Su\text{"}], address: \text{"}257\ 9th\ Av\ SW\text{"}]$

where $prolog, db, bsu$ are object identifiers.

The problem with this approach is that values such as integers, tuples, and sets etc. cannot be treated as first class citizens as we can only put them in the value part of an object. For a mathematical database, we may store integers and their assocated properties such as odd, even, prime, etc. In this case, an integer is better to be identified by itself rather than by a seperate identifier. The cases for tuples and sets are the same. This problem can be solved by treating everything as an object and defining attributes as mappings from objects to objects. As a result, whether or not to use an object identifier or a value for an object becomes a database design issue. We use the former if and only we want to support object sharing, cyclic object modeling and effective update management.

Objects often share common attributes. In object-oriented databases, similar objects are grouped together into classes which prescribe the attributes applicable to their instances. Classes are organized into subclass hierarchies, where instances of subclasses are instances of superclasses which is called instance inheritance, and prescribed attributes of superclasses can be inherited by the subclasses which is called property inheritance. Property inheritance enables us to reduce the redundancy of the specification while maintaining its completeness.

O-logic Object-oriented approach was first applied to deductive framework in O-logic [44]. The facts given in Example 3.2 can be represented in O-logic as follows:

$$\begin{array}{l} prolog : book[name \rightarrow \text{“}Prolog\text{”}, author \rightarrow bsu] \\ db : book[name \rightarrow \text{“}Databases\text{”}, author \rightarrow bsu] \\ bsu : person[firstname \rightarrow \text{“}Bob\text{”}, lastname \rightarrow \text{“}Su\text{”}, address \rightarrow \text{“}257\ 9th\ Av\ SW\text{”}] \end{array}$$

where $prolog, db, bsu$ are object identifiers and $book, person$ are classes.

However, tuples and sets are not allowed in O-logic. The objects can have arbitrary attributes but no structural information can be defined or declared on their classes. As rules allow us to generate new objects, a fundemental problem is how to automatically generate object identifiers. This issue is not properly addressed in O-logic.

Example 3.3 Consider the following rule in O-logic:

$$\begin{array}{l} P : ip[employee \rightarrow E, manager \rightarrow M] :- \\ \qquad E : employee[name \rightarrow M : string, works \rightarrow D] \\ \qquad D : dept[manager \rightarrow M : employee[name \rightarrow N] \end{array}$$

The intended meaning of this rule is that a pair employee-manage is interesting if the employee's department's manager's name coincides with the employee's name. However, it is not clear how to create an object identifier for each interesting pair in O-logic as discussed in [34].

F-logic Based on O-logic, an extended O-logic was first presented in [34], followed by a more general F-logic (frame logic) in [33] as a solution to the problems of O-logic. F-logic supports object identifiers, functor objects, sets, and subclass hierarchy. It also allows the declarations of attributes on classes.

Example 3.4 Following is an example of F-logic program:

$$
\begin{aligned}
&employee :: person\\
&sally : employee\\
&john : person\\
&mary : person\\
&25 : integer\\
&\text{"}Sally\text{"} : string\\
&person[name \Rightarrow string, age \Rightarrow integer, parents \Rrightarrow person]\\
&sally[name \rightarrow \text{"}Sally\text{"}, age \rightarrow 25, parents \twoheadrightarrow \{john, mary\}]
\end{aligned}
$$

where $person, employee$ and $string$ are classes while $sally, john$, and $mary$ are object identifiers. The symbols "::" and ":" are used to denote subclass relationship and class membership respectively. The statement $employee :: person$ says that $employee$ is a subclass of $person$; $sally : person$ says that $sally$ is an instance of the class $person$, etc. Attributes applicable to the class $person$ are declared using double-shafted arrows $\Rightarrow$ and $\Rrightarrow$ where the former is for scalar attributes and the latter for set-valued attributes. Attribute values of the object $sally$ are represented 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 allows the use of functor objects as generated object identifiers. For example, the interesting pair rule of O-logic in Example 3.3 can be represented in F-logic as follows:

$$
\begin{aligned}
&f(E, M) : ip[employee \rightarrow E, manager \rightarrow M] \;{:}{-}\\
&\qquad E : employee[name \rightarrow M : string, works \rightarrow D]\\
&\qquad D : dept[manager \rightarrow M : employee[name \rightarrow N]
\end{aligned}
$$

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

However, there are several problems with F-logic. First, there isn't a distinction between classes and objects as the domains for classes and objects are the same. They can be distinguished only from their occurrences in the program statements. The attribute declaration information cannot really be strictly seperated from attribute value information in F-logic. As a result, there is no clear seperation between the notions of schema and instance which is essential for database systems, see next section for details.

In database applications, a subclass may need to override attribute declarations inherited from superclasses in object-oriented paradigm. It may have to block the property inheritance as some attribute declarations meaningful for superclasses may not be meaningful for subclasses. They are not supported in F-logic.

Example 3.5 Consider an international corporation employee information database. We have three classes *employee*, *us_employee*, *cn_employee* (Canadian) so that *us_employee* and *cn_employee* are subclasses of *employee*. We may define the attribute *zipcode* of *employee* on *integer* which is fine for instances of *us_employee*. However, the *zipcode* for *cn_employee* must be defined on *string*. Therefore *cn_employee* should override the inherited attribute declaration of *zipcode* from *employee* which is impossible in F-logic.

Example 3.6 Consider 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 *age*, *father* and *mother* for *person*, then it is meaningful for *french* to inherit all attribute declarations from *person*, but not meaningful for *orphan* and *french_orphan* to inherit *father* and *mother* attribute. In other words, classes *orphan* and *french_orphan* should block the inheritance of attribute *father* and *mother* from *person*. Unfortunately, there is no mechanism in F-logic to block such inheritance.

In F-logic, sets are not treated as first-class citizens and the treatment of sets is too special. For example, values, object identifiers, and functor objects are identified by themselves but sets are not; values, object identifiers, and functor objects can have attributes but sets cannot. For a statement $bob[age \rightarrow 30]$, 30 is the value of the attribute *age* of *bob*. If sets are 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 of the 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-value attribute to be inferred in a number of steps using several rules in the way similar to COL. Following is an example:

$$X[ancestors \twoheadrightarrow \{Y\}] :- X[parents \twoheadrightarrow \{Y\}]$$
$$X[ancestors \twoheadrightarrow \{Y\}] :- X[parents \twoheadrightarrow \{Z\}], Z[ancetors \twoheadrightarrow \{Y\}]$$

However, this only special treatment of sets makes it impossible to specify the complete value of a set-value attribute. For example, if we know that *john* and *mary* are the two parents of *sally*, we can only use the statement:

$$sally[parents \twoheadrightarrow \{john, mary\}$$

which won't prevent additional persons from being inferred as *sally*'s parent.

ROL ROL [41] is another object-oriented deductive language which is based on F-logic but solves all the problems of F-logic discussed above.

In ROL, objects and classes are strictly distinguished and function at two different levels: instance and schema. In schema we declare classes, attributes on classes, and subclasses. The attribute declarations of a class constrain the instances of the class. Immediate subclasses are declared with *isa* relationship while general subclasses relationship $isa*$ are inferred based on the *isa* relationship. For example, *employee isa person* declares *employee* as an immediate subclass of *person* in ROL. Subclasses in ROL can inherits attribute declarations of their superclasses and can also override and block such inheritance too.

The database application considered in Example 3.5 can have the following schema in ROL:

$$employee[..., zipcode \Rightarrow integer]$$

$$us\_employee\ isa\ employee$$
$$cn\_employee[zipcode \Rightarrow string]\ isa\ employee$$

For the application considered in Example 3.6, we can use the following schema in ROL:

$$person[father \Rightarrow person, mother \Rightarrow person, age \Rightarrow integer]$$
$$orphan[father \Rightarrow none, mother \Rightarrow none]\ isa\ person$$
$$french\ isa\ person$$
$$french\_orphan\ isa\ french, orphan$$

Unlike F-logic, functors in ROL are treated as classes, functor objects are instances of the corresponding functor classes, and functor objects can embed functor objects. For example, functor objects $person(john), list(1, list(2, list(3, nil)))$ are instances of the functor class $person$, $list(integer, list)$ respectively. The interesting pair rule in Example 3.3 can be represented in ROL as follows:

$$\begin{aligned} ip(E, M)[employee \rightarrow E, manager \rightarrow M] \ {:}{-}& \\ & E : employee[name \rightarrow M, works \rightarrow D], \\ & D : dept[manager \rightarrow M], \\ & M : employee[name \rightarrow N] \end{aligned}$$

This rule generates functor objects such as $ip(bob, sam)$ as instances of the functor class ip and their attribute values.

Because of this treatment of functor objects, ROL subsumes Datalog, Prolog and LDL rules without grouping as special cases. For example, following LDL rules are also ROL rules:

$$int(s(X)) \ {:}{-}\ int(X)$$
$$ancestor(X, Y) \ {:}{-}\ parent(X, Y)$$
$$ancestor(X, Y) \ {:}{-}\ parent(X, Z), ancestor(Z, Y)$$
$$\begin{aligned} book\_deal(\{X, Y, Z\}) : -book(X, Px), book(Y, Py), book(Z, Pz) \\ X \neq Y, X \neq Z, Y \neq Z, Px + Py + Pz < 100 \end{aligned}$$

In ROL, sets are treated as first class citizen as values, object identifiers, and functor objects so that sets can also have attribute. For example, we can have the following fact in ROL:

$$\{john, smith\}[count \rightarrow 2]$$

ROL allows both partial and complete information about set-valued attribute to be specified by supporting partial and complete set terms as in Relationlog. Ground partial and complete set terms are called partial and complete sets respectively. For example $\langle tom \rangle$ and $\{smith, mary\}$ are partial and complete sets respectively. They carry different meanings when used as attribute values. Consider the following facts in ROL:

$$mary[parents \rightarrow \langle tom \rangle]$$
$$john[parents \rightarrow \{smith, mary\}]$$

The first one says $mary$'s parents include tom while the second says $john$'s parents are $smith$ and $mary$. Therefore, the partial set $\langle tom \rangle$ gives partial information about $john$'s parent and the complete set $\{smith, mary\}$ gives complete information about $john$'s parents.

Obviously, ROL supports the special set treatment of F-logic with partial sets. The values of *ancestors* attribute can be derived with following ROL rules:

$$X[ancestors \rightarrow \langle Y \rangle] :- X[parents \rightarrow \langle Y \rangle]$$
$$X[ancestors \rightarrow \langle Y \rangle] :- X[parents \rightarrow \langle Y \rangle], Y[ancestors \rightarrow \langle Z \rangle]$$

4 Schema and Higher-Order Features

A database usually has a schema. The schema provides the description of the database structure, constrains data in the database and query results and guarantee the consistency of the database. It is the basis for storage structure and query optimization strategies. Its importance is similar to the type declarations of programming languages. Unfortunately, the notion of schema is not supported in most deductive database languages, because they are based on the logic programming language Prolog which is inherently not typed. As a result, the semantics of schema, is not well-defined in most deductive frameworks. The current research on deductive databases has mainly focused on the evaluation strategies and the implemented deductive database system prototypes are mainly memory based database systems rather than true disk based ones.

We argue that schema will be the key component of any true deductive database. In such a case, the semantics of schema must be defined first. In all commercial relational database systems, such as Oracle, Ingres, Sybase, SQL is the uniform language used to define and manipulate both schema and data. However, SQL is by no means a deductive database language.

In order to reason about schematic information we need the capabilities of higher-order logics. But higher-order logics have been met with skepticism since the unification problem is undecidable. Normally, a non-logic-based language is provided to specify and manipulate the schema information.

In this section, we discuss the related problems and typical attempts that have been made in this respect.

Example 4.1 Consider the database consisting of three relations in Datalog: *hardware, software*, and *admin*, each of which represents the employees in the respective departments. The tuples in all three relations have attributes *name, phone* and *salary.* If we are allowed to use predicate variables, we can use the following query to list all predicates with arity 3 in the database:

$$?\!-\ Y(\_, \_, \_)$$

The answers to this query should be *hardware, software*, and *admin.*

Based on the above examples, we can see that we cannot query all relation names with higher-order variables in Datalog as we have to deal with arity of each relation. Furthermore, we cannot query attributes by using a query ?– $hardware(A, B, C)$ as the results to this query are attribute values of tuples.

L^2 L^2 is a higher-order language for deductive databases proposed in [37]. The problems discussed in Example 4.1 are solved by introducing attributes into tuples and redefining the database as a tuple.

Example 4.2 The database discussed in Example 4.1 can be defined in L^2 as

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

where *hardware*, *software* and *admin* are relation names (also attribute 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 $(name : ..., phone : ..., salary : ...)$. A specific attribute value *smith* in the relation *hardware* can be referenced with $.hardware(.name = smith)$.

The solution to higher-order unification is based on a bottom up semantics where unification is replaced by matching, i.e. only one of the two terms contains variables. The higher-order variables are limited to range over database attributes and predicates. A rule with higher-order variables can be rewritten by replacing variables with attributes or predicates. The rewritten rules are in first-order logic and their meaning is well-defined. Since the number of database attributes and predicates have to be finite, this makes the language decidable. This semantics is called *replacement semantics* in that paper. This approach provides a natural and simple way towards the integration of the definition and manipulation of schema and data.

Example 4.3 Consider the L^2 database in Example 4.2. We can define the following predicates (plain relations) with higher-order queries:

$$\begin{array}{l}
p(X) \coloneq .X \\
q(X) \coloneq .hardware(.X) \\
r(X) \coloneq .Y(.name = X) \\
s(X,Y) \coloneq .Y(.name = X) \\
t(X,Y) \coloneq .R1(.name = X, .phone = Z), .R2(.name = Y, .phone = Z) \\
\qquad\qquad\quad X \neq Z, Y1 \neq Y2
\end{array}$$

The first relation p contains all the relation names in the database. The second relation q contains all the attribute names of the relation *hardware* in the database. The third relation r contains all the names of the employees in *hardware, software* and *admin* relations. The fourth relation s contains all the names of the employees and their corresponding departments. The last relation t contains all the pair of names of employees who share their phone.

However, L^2 doesn't have a notion of schema. Consider the database in Example 4.2, if one of the relations r_1, r_2, r_3 is empty, then we don't have any structure information such as *name*, *phone*, and *salary* about it. Besides, it is not required that tuples in a relation have the same form. For example r_1 can contain two tuples: $(name : smith, phone : 12345, salary : 30k)$ and $(name : john, wife : mary)$. As a result, it is not possible to check whether or not the rules in L^2 are properly typed.

F-logic In object-oriented databases, we have class/subclass hierarchy and inheritance. A class corresponds to a relation. A natural question is how to reason about class/subclass hierarchy, attribute declarations, and class and attribute information of objects. F-Logic [33] is the first object-oriented language supporting modeling such higher-order features inferring schema information from the database. In F-logic, we can use variables for classes and attributes.

Example 4.4 Consider Example 3.4 again, we can use the following rules with higher-order queries to collect all classes and all attributes of the class *person* in F-logic:

$$
\begin{array}{l}
db[classes \twoheadrightarrow X] :- X :: Y \\
db[classes \twoheadrightarrow X] :- Y :: X \\
db[classes \twoheadrightarrow X] :- Y : X \\
db[classes \twoheadrightarrow X] :- X[A \Rrightarrow B] \\
db[classes \twoheadrightarrow X] :- X[A \Rightarrow B] \\
person[all\_attributes \twoheadrightarrow X] :- person[X \Rightarrow Y] \\
person[all\_attributes \twoheadrightarrow X] :- person[X \Rrightarrow Y]
\end{array}
$$

As shown in the above example, it is not very convenient to query all classes in F-logic. The difficulty is because F-logic does not distinguish between individual objects and classes. As a result, we have to query all possible places for classes. The distinction of two kinds of arrows is somewhat burdensome too.

Unlike L^2, F-logic has a notion of type checking for attribute values of objects with respect to their class attribute declarations. However, there is no notion of schema as objects and classes are mixed together.

ROL In ROL [41], there is a clear notion of schema. The semantics of higher-order features of ROL is based on replacement semantics of L^2 so that the language is still decidable.

Example 4.5 Consider the following ROL rules with higher-order queries:

$$
\begin{array}{l}
p(X) :- X[\,] \\
q(X) :- person[X \Rightarrow Y], A \neq age \\
r(X, A) :- X[A \Rightarrow integer] \\
s(X, L, Y) :- sally : X[L \rightarrow Y] \\
t(X) :- workstudent\ isa\ X \\
u(X) :- X\ isa*\ person, not\ X\ isa\ person
\end{array}
$$

The first rule collects all classes in the database. The second collects all attributes of *person* other than *age*. The third collects the classes and attributes that are defined using *integer*. The fourth collects information about *sally*'s class, attributes, and corresponding values. The fifth collects all immediate superclasses of the class *workstudent*. The last collects all non-immediate subclasses of *person*.

5 Updates

In Prolog, the basic update primitives are *assert* and *retract*. *Assert* is used to insert a single clause into the database. Assert always succeeds initially and fails when the computation backtracks. Clauses are deleted from the database in Prolog by calling *retract*. Initially, retract deletes the first clause in the database which unifies with the argument of retract. On backtracking, the next matching clause is removed. Retract fails when there are no remaining matching clauses.

The semantics of assert and retract are not well defined. The exact effect of calling code containing assert and retract is often difficult to predict. The resulting database update relies on the procedural semantics of Prolog, rather than just the declarative semantics.

Example 5.1 Suppose we have a database consisting of a single predicate *employee* with arity two. The first argument is the employee's name and the second is the employee's salary. The following rule can be used to give every employee a $X\%$ salary increase in Prolog:

$$\begin{aligned} raise(X) :- \ & retract(employee(N, S_1)), \\ & Y = 1 + X, S_2 = S_1 * Y, \\ & assert(employee(N, S_2)), \\ & fail. \end{aligned}$$

If we want to give every employee a 10% salary increase, we can make a query $?\text{–}raise(0.1)$.

The problem with Prolog is that if we change the order of *assert* and *retract* in the body of a rule, we may get different results. The procedural semantics of Prolog affects what database updates are done. The order of execution of subgoals is as important as the logical content of the goal. 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. Even in the single user case, concurrent access is required to data structures on disk which may be quite complex. Multi-user access creates even more difficulties.

Datalog as a subset of Prolog is a pure declarative query language without any data manipulation part. How to incorporate them into deductive databases has been the focus of active research during the last few years and various approaches have been proposed [1, 9, 11, 10, 14, 20, 31, 36, 46, 49]. In this section, we discuss typical approaches.

Datalog/UT Datalog/UT [42] extends Datalog by incorporating update rules that are used to update extensional predicates. Every predicate in Datalog/UT is associated with a temporal argument called time point which is an integer. The updates sequence is represented implicitly by the temporal information associated with predicates. There are several advantages of using temporal information. First, time points are naturally linearized. Second, updates are naturally associated with times. Most importantly, the proposed update language has a simple and clear declarative semantics which is a natural and direct extension of the Datalog semantics and reduces to it in the special case. The database of Datalog/UT is a history database and updates are represented as changes with "logical" time.

Consider the example given in Example 5.1. The only predicate in the database is *employee* with arity two. The first argument is the name of the employee and the second is the salary. The last argument is the integer indicating the time. The following Datalog/UT program can increase every employee's salary by 10%:

$$latest\_salary(N,S)@T \text{ :- } employee(N,S)@T_0, T \geq T_0,$$
$$not(employee(N,\_)@T_1, T \geq T_1, T_1 > T_0).$$

$$employee(N,S_2)@T \text{ :- } raise(X)@T,$$
$$latest\_salary(N,S_1)@T, Y = 1 + X, S_2 = S_1 * Y.$$

The first rule specifies how to retrieve the "latest" version of each facts at some time T. It says that the latest version at time T can be obtained from the fact at some time T_0 earlier but closest to T. The second rule tells how to create a new version for the facts the salary arguments of which have been increased by X%. It first finds a latest version for the fact at a particular time specified by T and then asserts a new version of the fact.

In Datalog/UT, only addition of facts into the database is allowed. For example, if we want to give every employee a 10% salary increase, we just insert the fact $raise(0.1)$ at a certain time t. Deletion or modification is inferred using rules. This is similar to [54]. The database is thus a history database with temporal information and it is possible to query information about the past. This approach may be used just to express the semantics of updates and changes to the traditional snapshot database. Used in this way, time would have no meaning within the database itself. In such a system the state of the database would be at its current time. A query could be made only at the current time and updates would be inserted and occur at the current time. The appropriate sequencing of updates would be ensured by giving independent sources of updates (for example different users in a multi-terminal system) their own unique time stamps.

The main advantage of Datalog/UT is that it permits a declarative semantics based on a bottom-up, least-fixpoint computation.

Dynamic Logic The notion of states is inherent in any notion of updates. The dynamic logic approach assigns state transition semantics to a logic program [49]. The closure operator associated with a logic program P computes a state of P in the sense that it assigns valuations to the variable of P. Updates can be viewed as transitions of a state through a state-space. In the absence of updates, a classical logic program has only one state and queries map this state to itself. So it reduces to the classical semantics of logic program. Two kinds of updates are distinguished there which have different semantics. First, those in which update actions depend on the order of execution, that is, different orders of execution may yield different final states. This kind of update is represented by $(\alpha;\beta)$ where α and β stand for update predicates. The semantics for this kind does not require that α executed before or after β gives the same result. The other kind are those in which all different orders of execution yield the same final state. This kind of update predicates are represented by α, β. For example, the update rule in Example 5.1 can be represented as follows:

$$raise(X) \text{ :- } employee(E,S_1),$$

$$Y = 1 + X, S_2 = S_1 * Y,$$
$$(-employee(E, S_1); +employee(E, S_2)).$$

where $employee(X, S_1)$ is a query which generates substitutions for X and S_1, $-employee(X, S_1)$ means to delete substituted predicate $employee(X, S_1)$, while $+employee(X, S_2)$ means to insert substituted predicate $employee(X, S_2)$. $(-employee(X, S_1); +employee(X, S_2))$ is used to change the salary of X from S_1 to S_2.

The deletion and insertion operations can only occur in queries or in the body of a rule and the intended updates can be done only by querying the head of the rule (possibly several times), when they are in the body of a rule. The dynamic logic interpretation of updates [49]. gives a semantics which is consonant with the operational meanings of the update predicates. The main problem with this approach is that this semantics is not declarative.

Transactions The notion of transaction has been widely used in relational database systems. In order to reduce the number of database states by grouping small changes into a big one and to address the issue of concurrency and atomicity of certain operations, the notion of transaction is introduced into deductive databases in [48]. A transaction is a collection of updates which must be done atomically. This naturally specifies some form of concurrency control. A transaction is specified by two sets: the facts to be deleted (D) and the facts to be inserted (I). The new database state (New_db) after the transaction is defined in terms of the old database state (Old_db) before the transaction, D and I:

$$New\_db = (Old\_db - D) \cup I$$

This definition corresponds to performing deletions before insertions. Only if the transaction is committed, then the updates have been made by first doing all the deletions then all the insertions. Obviously, consistent database updates could be achieved by using a transaction primitive for Prolog. The proposed syntax of a transaction primitive in [48] is

all Local_vars transaction Goal

where *Local_vars* is a term containing variables local to *Goal*, *Goal* is any Prolog goal and may contain "calls" to the procedure *db_delete* and *db_insert*.

Using a transaction, we can increase the salary of every employee by 10% as follows:

$$\begin{aligned} all\ S_1, S_2\ transaction\ (&employee(X, S_1),\\ &S_2 = S_1 * 1.1,\\ &db\_delete(employee(X, S_1)),\\ &db\_insert(employee(X, S_2))). \end{aligned}$$

Because *db_insert* and *db_delete* always succeed and only change the database at the end of the transaction, the declarative semantics for the solutions to the goal is straightforward. The main advantage of introducing transactions is that it gives a simple declarative semantics for updates.

Transaction Logic The ideas of using transaction have been formalized in transaction logic $\mathcal{T}_\mathcal{R}$ [9]. $\mathcal{T}_\mathcal{R}$ is an extension of first-order logic, both syntactically and semantically. It generalizes the above two approaches. It has a natural model theory and a sound and complete proof theory. Two operations are provided to delete and insert facts: *del* and *ins*. A new logical connective called *serial conjunction* denoted by $\otimes$ is introduced to sequence transactions, where $\alpha \otimes \beta$ means do α then do β. The updates given in Example 5.1 can be represented as a transaction which is a rule in transaction logic as follows:

$$\begin{aligned} raise(X) \coloneq\ & (employee(E, S_1), Y = X + 1) \otimes \\ & del : employee(E, S_1) \otimes ins : employee(E, S_1 * Y). \end{aligned}$$

In $\mathcal{T}_\mathcal{R}$, a rule can have other transactions so that nested transactions are allowed. Unlike Prolog, the main novel feature of $\mathcal{T}_\mathcal{R}$ is that it has a declarative semantics.

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 deductive power. The need for expressiveness has forced the deductive database languages away from their simple roots in logic programming and traditional relational data model.

In this paper, we have discussed four broad areas where problems are apparent in deductive databases. In each case we have examined several solutions. It seems that a synthesis using transaction logic to represent updates, and an extension of ROL and F-logic family to represent object identity, complex object, inheritance with overriding and blocking, and schema is a viable deductive database language. Clearly, more work is needed to determine whether the resulting language is semantically sound, sufficiently expressive and efficiently implementable.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] S. Abiteboul. Updates, a new database frontier. In *Proc. Intl. Conf. on Data Base Theory*, pages 1–18, Pruges, Belgium, 1988. Springer-Verlag LNCS 326.

[2] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *J. Computer and System Sciences*, 33(3):361–393, December 1986.

[3] S. Abiteboul and S. Grumbach. COL: A logic-based language for complex objects. *ACM TODS*, 16(1):1–30, 1991.

[4] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Trans. on Database Systems*, 10(2):230–260, June 1985.

[5] F. Bancilhon, F. D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. ACM Sym. on Principles of Database Systems*, pages 1–16, Cambridge, Massachusetts, 1986.

[6] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 16–52, Washington, D.C., 1986.

[7] C. Beeri. Formal models for object-oriented databases. In W. Kim, J.M. Nicolas, and S. Nishio, editors, *Deductive and Object-Oriented Databases*, pages 405–430, Kyoto, Japan, 1989. North-Holland.

[8] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set construction in a logic database language. *J. Logic Programming*, 10(3,4):181–232, April/May 1991.

[9] A. J. Bonner and M. Kifer. Transaction logic programming. In *Proc. Intl. Conf. on Logic Programming*, pages 257–279, Budapest, Hungary, 1993. MIT Press.

[10] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, October 1994.

[11] A.J. Bonner, M. Kifer, and M. Consens. Database programming in transaction logic. In *Proc. Intl. Workshop on Database Programming Languages*, pages 309–337, Manhattan, New York City, 1993. Morgan-Kaufmann.

[12] M.L. Brodie. On the development of data models. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, pages 19–48. Springer-Verlag, 1984.

[13] M.L. Brodie and D. Ridjanovic. On the design and specification of database transactions. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, pages 232–276. Springer-Verlag, 1984.

[14] F. Bry. Intensional updates: Abduction via deduction. In *Proc. Intl. Conf. on Logic Programming*, pages 561–575, Budapest, Hungary, 1990. MIT Press.

[15] M. Carey, D. DeWitt, and S. Vanderberg. A data model and query language for EXODUS. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 413–423, Chicago, Illinois, 1988.

[16] S. Ceri, G. Gottlob, and T. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[17] P. P. S. Chen. The entity-relationship model. *ACM Trans. on Database Systems*, 1(1):9–36, March 1976.

[18] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):76–90, March 1990.

[19] L. Colby. A recursive algebra and query optimization for nested relations. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 124–138, Portland, Oregon, 1989.

[20] C. de Maindreville and E. Simon. Modelling non deterministic queries and updates in deductive databases. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 395–406, Los Angles, California, USA, 1988. Morgan Kaufmann Publishers, Inc.

[21] M. Derr, S. Morishita, and G. Phipps. Design and implementation of the Glue-Nail database system. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 147–167, Washington, D.C., 1993.

[22] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):35–48, January 1991.

[23] D. H. Fishman, B. B., H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An object-oriented database management system. *ACM Trans. on Office Information Systems*, 5(1):48–69, January 1987.

[24] H. Gallaire, J. Minker, and J. M. Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–186, June 1984.

[25] M. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Trans. Database Systems*, 6(3):351–386, September 1981.

[26] J. Han, L. Liu, and Z. Xie. LogicBase: A deductive database system prototype. In *Proc. Intl. Conf. on Information and Knowledge Management*, pages 226–233, Gaithersburg, Maryland, 1994. ACM.

[27] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[28] Y. E. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 382–394, Los Angeles, California, USA, 1988. Morgan Kaufmann Publishers, Inc.

[29] H. Ishikawa, F. Suzuki, F. Kozakura, A. Makinouchi, M. Miyagishima, Y. Izumida, M. Aoshima, and Y. Yamane. The model, language, and implementation of an object-oriented multimedia knowledge base management system. *ACM TODS*, 18(1):1–50, 1993.

[30] B. Jiang. A suitable algorithm for computing partial transitive closures. In *Proc. Intl. Conf. on Data Engineering*, pages 264–271, Kobe, Japan, 1990. IEEE Computer Society.

[31] A. C. Kakas and P. Mancarella. Database updates through abduction. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 650–661, Brisbane, Queensland, Australia, 1990. Morgan Kaufmann Publishers, Inc.

[32] S.N. Khoshafian and G. P. Copeland. Object identity. In *OOPSLA '86 Proceedings*, pages 406–416. ACM New York, 1986.

[33] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42(4):741–843, July 1995.

[34] M. Kifer and J. Wu. A logic for programming with complex objects. *J. Computer and System Sciences*, 47(1):77–120, August 1993.

[35] W. Kim. Object-oriented databases: Definition and research direction. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):327–341, September 1990.

[36] M. Kramer, G. Lausen, and G. Saake. Updates in a rule-based language for objects. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 225–236, Vancouver, British Columbia, Canada, 1992. Morgan Kaufmann Publishers, Inc.

[37] R. Krishnamurthy and S. Naqvi. Towards a real horn clause language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 252–263, Los Angles, California, USA, 1988. Morgan Kaufmann Publishers, Inc.

[38] G. M. Kuper. Logic programming with sets. *J. Computer and System Sciences*, 41(1):44–64, August 1990.

[39] C. Lecluse and P. Richard. The O_2 database programming language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 411–422, Amsterdam, The Netherlands, 1989. Morgan Kaufmann Publishers, Inc.

[40] M. Liu. HLO: A higher-order deductive object-oriented database language. *Informatica: An International Journal of Computing and Information*, 19(3):319–331, 1995.

[41] M. Liu. The LOL deductive database programming language. In *Proc. Intl. Symp. on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, pages 454 – 468, Aachen, Germany, September 25-27 1996. Springer-Verlag LNCS 1140.

[42] M. Liu and J. Cleary. Declarative updates in deductive databases. *Journal of Computing and Information (Special Issue: Proceedings of the 6th International Conference on Computing and Information)*, 1(1):1435–1446, 1994.

[43] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2 edition, 1987.

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

[45] D. Maier. Why database languages are a bad idea. In *Proc. Workshop on Database Programming Languages*, Roscoff, France, 1987.

[46] S. Manchanda. Declarative expression of deductive database updates. In *ACM Symp. on Principles of Database Systems*, pages 93–100, Philadelphia, Pennsylvania, 1989.

[47] K. Morris, J. D. Ullman, , and A. V. Gelder. Design overview of the Nail! system. In *Proc. Intl. Conf. on Logic Programming*, pages 554–568, London, England, 1986. MIT Press.

[48] L. Naish, L. A. Thom, and K. Ramamohanarao. Concurrent database updates in prolog. In *Proc. Intl. Conf. on Logic Programming*, pages 178–189, Melbourne, Victoria, 1987. MIT Press.

[49] S. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proc. ACM Syum. on Principles of Database Systems*, pages 261–272, Austin, Texas, 1988.

[50] J. Peckham and F. Maryanski. Semantic database models. *ACM Computing Surveys*, 20(3):153–189, September 1988.

[51] R. Ramakrishman, editor. *Applications of Logic Databases.* Kluwer Academic Publishers, 1994.

[52] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. XSB as an efficient deductive database engine. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 442–453, Minneapolis, Minnesota, 1994.

[53] R. Reiter. Towards a logical reconstruction of relational database theory. In M.L. Brodie, J. Mylopoulos, and J.W. Schmidt, editors, *On Conceptual Modelling*, pages 191–233. Springer-Verlag, 1984.

[54] R. Reiter. On formalizing database updates: Preliminary report. In *Proc. Intl. Conf. on Extending Database Technology*, pages 10–20, Vienna, Austria, 1992. Springer-Verlag.

[55] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM TODS*, 13(4):389–417, 1988.

[56] S. Y. W. Su. Modeling integrated manufacturing data with SAM*. *IEEE Computer Society*, 19(1):34–49, January 1986.

[57] S. Tsur and C. Zaniolo. LDL: A logic-based data language. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 33–41, Kyoto, Japan, 1986. Morgan Kaufmann Publishers, Inc.

[58] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[59] J. Vahani, K. Ramamohanarao, K. Kemp, D. B. Somogyi, and P. J. Stucky. Design overview of the Aditi deductive database system. In *Proc. Intl. Conf. on Data Engineering*, pages 240–247, Kobe, Japan, 1991. IEEE Computer Society.