In Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming (DDLP '98), Manchester, UK, June 20, 1998, pp. 99-112.

Overview of Datalog Extensions with Tuples and Sets

Mengchi Liu
Department of Computer Science
University of Regina, Saskatchewan, Canada S4S 0A2
Email: mliu@cs.uregina.ca

Abstract

Datalog (with negation) is the most powerful query language for relational database with a well-defined declarative semantics based on the work in logic programming. However, Datalog only allows inexpressive flat structures and cannot directly support complex values such as nested tuples and sets common in novel database applications. For these reasons, Datalog has been extended in the past several years to incorporate tuple and set constructors. In this paper, we examine four different Datalog extensions: LDL, COL, Hilog and Relationlog.

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 ever increasingly complex data reliably in various memory devices. The underlying structure of databases has been the primary focus of research which leads to the development of data models. The most well-known and widely used data model is the relational data model [12]. 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.

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 [13, 29, 53], 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 [36].

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 [16, 17, 24, 38, 56]. Relational databases and logic programming have been found to be quite similar in their representation of data at the language level. They have also been found to be complementary in many aspects. Logic programming has contributed to the understanding of the semantics of a relational database, and has extended the expressive power of relational databases. On the other hand, relational database systems are superior to the standard implementations of Prolog with respect to data independence, secondary storage access, concurrency, recovery, security and integrity [54]. 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. However, the expressive power and functionality offered by the relational database query languages is 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* [40] 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 [47, 53].

The integration of logic programming and relational database techniques has led to the active research area of deductive databases [18, 8, 21]. It 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 [57, 8].

In the past few years, various set-oriented evaluation strategies specific for deductive databases have been the main focus of extensive research [4, 5, 7, 8, 23, 25, 42, 49, 57, 55] and a number of deductive database systems or prototypes based on Datalog have been developed and reported. These include Nail [41], LOLA [15], Glue-Nail [14], XSB [50], CORAL [46], Aditi [58], LogicBase [22], Declare/SDS [26] etc. See [45] for a survey of these deductive database systems.

However, deductive databases based on Datalog only provide inexpressive flat structures and cannot directly support complex values such as nested tuples and sets common in novel database applications. 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:

$$
\begin{array}{l}
car(vehicle(ny\text{-}858778,\ chevy\text{-}vega,\ 1983),\\
\quad owner(name(arthur, keller),\\
\quad\quad getFrom(dealer)))\\
car(vehicle(ny\text{-}358735,\ wolkswagen,\ 1990),\\
\quad owner(name(arthur, keller),\\
\quad\quad getFrom(owner(name(kurt, godel),\\
\quad\quad\quad getFrom(owner(name(norbert, wiener),\\
\quad\quad\quad\quad getFrom(dealer)))))))
\end{array}
$$

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

Prolog also indirectly supports sets by using lists. In Prolog, the 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 without any 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, $S = \{X \mid P\}$.

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 [31].

For these reasons, Datalog has been extended in the past several years to incorporate tuple and/or set constructors [1, 6, 9, 10, 30, 31, 35, 51]. In this paper, we examine four different Datalog extensions: LDL, COL, Hilog and Relationlog from a historical perspective in the subsequent sections.

There are also other significant extensions to Datalog which are beyond the scope of the paper. These include null values, aggregation, hypothetical reasoning, temporality, constraints, object-orientation, updates, etc.

2 LDL

LDL (Logical Data Language) [6, 44, 54] is the first language that extends Datalog to support complex values with well defined semantics. It has also been implemented as a main memory deductive database system at MCC [3, 11]. 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*:

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:

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

To support the 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. The following are several examples.

Example 2.2 If we want 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:

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

Note that we are able to 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:

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

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 has to 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 occurring in the program.

There are three problems in LDL. First, the grouping mechanism is restricted to a single rule rather than a set of rules (or 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 $parentsof(bob, \{pam, tom\})$:

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

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:

$$\begin{aligned} &ancestorsof(X, \langle Y \rangle) :- parentsof(X, S), Y \in S \\ &ancestorsof(X, \langle Y \rangle) :- parentsof(X, S_1), Z \in S_1, \\ &\qquad\qquad ancestorsof(Z, S_2), Y \in S_2 \end{aligned}$$

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 we 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:

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

Finally, it is cumbersome to access deeply nested data and to nest/unnest relations as we have to 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 into the level we want and introducing intermediate variables S_1 and S_2:

$$\begin{aligned} &?\!-dept\_employees(cpsc, S_1), empl(X, S_2) \in S_1, db \in S_2. \\ &?\!-dept\_employees(\_, S_1), empl(\_, S_2) \in S_1, ai \in S_2 \end{aligned}$$

Example 2.7 The nested relation *Dept_Employees* in Example 2.1 may be obtained from the following normalized relation *Dept_Employee_Area* by using the nest operation of extended relational algebra defined in [48] 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:

$$\begin{array}{l} temp_1(D, E, \langle A \rangle) \coloneq dept\_employee\_area(D, E, A) \\ temp_2(D, empl(E, As)) \coloneq temp_1(D, E, As) \\ dept\_employees(D, \langle Es \rangle) \coloneq temp_2(D, Es) \end{array}$$

3 COL

COL (Complex Object Language) [1] 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 may be represented in COL as follows:

$$\begin{array}{l} dept\_employees(cpsc, \{[bob, \{db, ai\}], [joe, \{os, pl, db\}]\}) \\ dept\_employees(math, \{[sam, \{gt, cm, si\}], [tom, \{ca\}]\}) \end{array}$$

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 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:

$$\begin{array}{l} par(X) \ni Y \coloneq fatherof(X, Y) \\ par(X) \ni Y \coloneq motherof(X, Y) \\ parentsof(X, par(X)) \end{array}$$

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

$$\begin{array}{l} anc(X) \ni Y \coloneq par(X) \ni Y \\ anc(X) \ni Y \coloneq par(X) \ni Z, anc(Z) \ni Y \\ ancestors(X, anc(X)) \end{array}$$

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

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

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

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

$$?\text{–}dept\_employees(cpsc, f), f \ni [X, g], g \ni db$$
$$?\text{–}dept\_employees(\_, f), f \ni [\_, g], g \ni ai$$

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

$$f(D, E) \ni A :\text{–}\ dept\_employee\_area(D, E, A)$$
$$temp(D, E, f(D, E))$$
$$g(D) \ni [E, As] :\text{–}\ temp(D, E, As)$$
$$dept\_employees(D, g(D))$$

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 $A :\text{–}\ L_1, ..., L_n$, where $n \geq 0$, 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 3.1 in COL:

$$anc(X) \ni Y \Leftarrow parents(X, S_1), S_1 \ni Z, ancestors(Z, S_2), S_2 \ni Y.$$
$$ancestors(X, anc(X)).$$

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

The stratification of COL has the following problem.

Example 3.2 [Adapted from [28]] Consider the following program in COL:

$$person(peter, \{bridge\})$$
$$person(tom, \{chess, tennis\})$$
$$person(tom, hobby(peter))$$
$$hobby(X) \ni Y :\text{–}\ person(X, S), S \ni Y$$

The intended meaning for 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.

4 Hilog

Hilog [9, 10] is a typed extension of Datalog that intends to solve part of the problems of LDL. In Hilog, every set or tuple must be associated with a name which 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:

$$\begin{aligned} dept\_employees\{ & dept(cpsc,\ employees\{employee(bob, areas\{db, ai\}), \\ & \qquad\qquad\qquad\qquad employee(joe, areas\{os, pl, db\})\}), \\ & dept(math, employees\{employee(sam, areas\{gt, cm, si\}), \\ & \qquad\qquad\qquad\qquad employee(tom, areas\{ca\})\})\} \end{aligned}$$

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 it 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 by 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:

$$?\text{–}dept(cpsc, employees\{employee(X, areas\{db\})\})$$
$$?\text{–}dept\_employees\{dept(\_, employees\{employee(\_, areas\{ai\})\})\}$$

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:

$$?\text{–}areas\{ai\}$$

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:

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

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

$$parentsof(X, parents\{Y\}) :\text{–}\ fatherof(X, Y)$$
$$parentsof(X, parents\{Y\}) :\text{–}\ motherof(X, Y)$$

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

$$ancestorsof(X, ans\{Y\}) :\text{–}\ parentsof(X, par\{Y\})$$
$$ancestorsof(X, ans\{Y\}) :\text{–}\ parentsof(X, par\{Z\}), ancestorsof(Z, ans\{Y\})$$

The unstratified COL program in Example 3.2 can be expressed in Hilog as follows:

$$person(peter, hobbies\{bridge\})$$
$$person(tom, hobbies\{chess, tennis\})$$
$$person(tom, hobbies\{X\}) :\text{–}\ person(peter, hobbies\{X\})$$

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

5 Relationlog

Relationlog (Relation LOGic) [32, 35] is another typed extension of Datalog with powerful set and tuple constructors. It has been implemented as a disk-resident deductive database system at the University of Regina [52]. It integrates 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 the same as in COL.

Relationlog allows negation and supports Hilog's special set treatment with partial set terms and eliminates the unique name assumption. A partial set term is of 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 of 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 of 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 accumulate or group partial information for the corresponding set as in LDL and 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:

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

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 $A :- L_1, ..., L_n$ 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 heads have the same predicate symbol as L_i, 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 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:

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

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:

$$?\text{–}dept\_employees(cpsc, \langle [X, \langle db\rangle])$$
$$?\text{–}dept\_employees(\_, \langle [\_, \langle ai\rangle]\rangle)$$

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:

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

The unstratified COL program in Example 3.2 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 Tom's father is unknown, cannot be represented in Relationlog. Therefore, it is not clear what the meaning of recursive rules is 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 not an easy way to query sets of some cardinality in Relationlog, such as sets of cardinality ≥ 2 or ≤ 2.

6 Conclusion

In this paper, we have examined four Datalog extensions: LDL, COL, Hilog, and Relationlog. LDL is an untyped Datalog extension that supports sets directly and tuples indirectly. It provides powerful set enumeration and set grouping mechanisms to manipulate sets. COL is a typed Datalog extension that directly supports sets and tuples. It introduces data functions to represent functional dependencies and to manipulate sets, especially to group sets with several rules recursively. Hilog is also a typed Datalog extension that directly supports sets and tuples. It gives a special meaning to sets so that direct access to deeply nested data and grouping using several rules recursively are allowed. Relationlog is another typed Datalog extension that directly supports sets and tuples. Its partial set terms and complete set terms generalize set grouping and set enumeration mechanisms of LDL. Besides, its partial set terms incorporate the special treatment of sets in Hilog so that direct access to deeply nested data and grouping using several rules recursively are allowed as well.

However, these languages are all value-based. They are lacking in its data modeling power compared to deductive object-oriented database languages [2, 27, 28, 33, 34, 37, 39, 43, 59]. Besides, none of them are computationally complete so that all programming tasks can be performed in a single framework. They do not allow unknown values which are common in database applications. Finally, they require stratification with regard to sets and negation. Indeed, it is still an open problem whether novel logic programming semantics such as stable model semantics [20] and well-founded semantics [19] can be incorporated into these languages without sacrificing their important features.

Acknowledgments

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

References

[1] S. Abiteboul and S. Grumbach. COL: A Logic-Based Language for Complex Objects. *ACM TODS*, 16(1):1–30, 1991.

[2] S. Abiteboul and P.C. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.

[3] Natraj Arni and KayLiang Ong. LDL User's Guide, Edition 2.0. Technical Report Carnot-012-93(P), MCC, 1993.

[4] F. Bancilhon, F.D. Maier, Y. Sagiv, and J.D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 1–16, Cambridge, Massachusetts, 1986.

[5] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., 1986.

[6] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set Construction in a Logic Database Language. *J. Logic Programming*, 10(3,4):181–232, 1991.

[7] C. Beeri and R. Ramakrishnan. On the Power of Magic. *J. Logic Programming*, 10(3,4):255–299, 1991.

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

[9] Q. Chen and W. Chu. HILOG: A High-Order Logic Programming Language for Non-1NF Deductive Databases. In W. Kim, J.M. Nicolas, and S. Nishio, editors, *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 431–452, Kyoto, Japan, 1989. North-Holland.

[10] Q. Chen and Y. Kambayashi. Nested Relation Based Database Knowledge Representation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 328–337, Denver, Colorado, 1991.

[11] 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, 1990.

[12] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Comm. ACM*, 13(6):377–397, 1970.

[13] Alain Colmerauer. Prolog in 10 Figures. *Communications of the ACM*, 28(12):1296–1310, 1985.

[14] M. Derr, S. Morishita, and G. Phipps. Design and Implementation of the Glue-Nail Database System. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 147–167, Washington, D.C., 1993.

[15] B. Freitag, H. Schutz, and G. Specht. LOLA – A Logic Language for Deductive Databases and its Implementation. In *Proceedings of the International Symposium on Database Systems for Advanced Applications (DASFAA '91)*, pages 216–225, Tokyo, Japan, 1991.

[16] H. Gallaire. Impacts of Logic on Data Bases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 248–259, Cannes, France, 1981. IEEE, New York.

[17] H. Gallaire and J. Minker, editors. *Logic and Data Bases.* Plenum, New York, 1978.

[18] H. Gallaire, J. Minker, and J. M. Nicolas. Logic and Databases: A Deductive Approach. *ACM Computing Surveys*, 16(2):153–186, 1984.

[19] A. V. Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of ACM*, 38(3):620–650, 1991.

[20] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the International Conference and Symposium on Logic Programming*, pages 1070–1080, Washington, USA, 1988. MIT Press.

[21] J. Grant and J. Minker. The Impact of Logic Programming on Databases. *Comm. ACM*, 35(3):67–81, 1992.

[22] J. Han, L. Liu, and Z. Xie. LogicBase: A Deductive Database System Prototype. In *Proceedings of the International Conference on Information and Knowledge Management*, pages 226–233, Gaithersburg, Maryland, 1994. ACM.

[23] Y.E. Ioannidis and R. Ramakrishnan. Efficient Transitive Closure Algorithms. In *Proceedings of the International Conference on Very Large Data Bases*, pages 382–394, Los Angeles, California, USA, 1988. Morgan Kaufmann Publishers, Inc.

[24] B.E. Jacobs. On Database Logic. *Journal of ACM*, 29(2):310–332, 1982.

[25] B. Jiang. A Suitable Algorithm for Computing Partial Transitive Closures. In *Proceedings of the International Conference on Data Engineering*, pages 264–271, Kobe, Japan, 1990. IEEE Computer Society.

[26] W. Kiebling, H. Schmidt, W. Straub, and G. Dunzinger. DECLARE and SDS: Early Efforts to Commercialize Deductive Database Technology. *VLDB Journal*, 3(2):211–243, 1994.

[27] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of ACM*, 42(4):741–843, 1995.

[28] M. Kifer and J. Wu. A Logic for Programming with Complex Objects. *J. Computer and System Sciences*, 47(1):77–120, 1993.

[29] Robert Kowalski. The Early Years of Logic Programming. *Communications of the ACM*, 31(1):38–43, 1988.

[30] R. Krishnamurthy and S. Naqvi. Towards a Real Horn Clause Language. In *Proceedings of the International Conference on Very Large Data Bases*, pages 252–263, Los Angeles, California, USA, 1988. Morgan Kaufmann Publishers, Inc.

[31] G. M. Kuper. Logic Programming with Sets. *J. Computer and System Sciences*, 41(1):44–64, 1990.

[32] M. Liu. Relationlog: A Typed Extension to Datalog with Sets and Tuples (Extended Abstract). In *Proceedings of the International Logic Programming Symposium (ILPS '95)*, pages 83–97, Portland, Oregon, U.S.A., December 4-7 1995. MIT Press.

[33] M. Liu. ROL: A Deductive Object Base Language. *Information Systems*, 21(5):431 – 457, 1996.

[34] M. Liu. An Overview of Rule-based Object Language. *Journal of Intelligent Information Systems*, 10(1):5–29, 1998.

[35] M. Liu. Relationlog: A Typed Extension to Datalog with Sets and Tuples. *Journal of Logic Programming*, 36(3):271–299, 1998.

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

[37] Y. Lou and M. Ozsoyoglu. LLO: A Deductive Language with Methods and Method Inheritance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 198–207, Denver, Colorado, 1991.

[38] D. Maier, editor. *The Theory of Relational Databases*. Computer Science Press, 1983.

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

[40] D. Maier. Why Database Languages are a Bad Idea. In *Proceedings of the Workshop on Database Programming Languages*, Roscoff, France, 1987.

[41] K. Morris, J.D. Ullman, , and A.V. Gelder. Design Overview of the Nail! System. In *Proceedings of the International Conference on Logic Programming*, pages 554–568, London, England, 1986. MIT Press.

[42] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishmnan. Magic Conditions. *ACM TODS*, 21(1):107–155, 1996.

[43] I.S. Mumick and K.A. Ross. Noodle: A Language for Declarative Querying in an Object-Oriented Database. In S. Ceri, K. Tanaka, and S. Tsur, editors, *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 360–378, Phoenix, Arizona, USA, 1993. Springer-Verlag LNCS 760.

[44] Shamim Naqvi and Shalom Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, 1989.

[45] R. Ramakrishman and J. D. Ullman. A Survey of Deductive Database Systems. *J. Logic Programming*, 23(2):125–150, 1995.

[46] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL Deductive System. *VLDB Journal*, 3(2):161–210, 1994.

[47] 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.

[48] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM TODS*, 13(4):389–417, 1988.

[49] D. Sacca and C. Zaniolo. Magic Counting Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–59, 1987.

[50] K. Sagonas, T. Swift, and D.S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 442–453, Minneapolis, Minnesota, 1994.

[51] P. Sander. Boolean lattices of nested relations as a foundation for rule-based database languages. *Data and Knowledge Engineering*, 8:93–130, 1992.

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

[53] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[54] S. Tsur and C. Zaniolo. LDL: A Logic-Based Data Language. In *Proceedings of the International Conference on Very Large Data Bases*, pages 33–41, Kyoto, Japan, 1986. Morgan Kaufmann Publishers, Inc.

[55] J. D. Ullman. Bottom-up Beats Top-Down for Datalog. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 140–149, Philadelphia, Pennsylvania, 1989.

[56] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

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

[58] J. Vaghani, K. Ramanohanarao, D.B. Kemp, Z. Somogyi, P.J. Stuckey, T.S. Leask, and J. Harland. The Aditi Deductive Database System. *VLDB Journal*, 3(2):245–288, 1994.

[59] C. Zaniolo. Object Identity and Inheritance in Deductive Database —an Evolutionary Approach. In W. Kim, J.M. Nicolas, and S. Nishio, editors, *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 7–21, Kyoto, Japan, 1989. North-Holland.