

Introduction to the Relationlog System

Riqiang Shan and Mengchi Liu
Department of Computer Science,
University of Regina
Saskatchewan, Canada S4S 0A2
Email: {shanriq, mliu}@cs.uregina.ca

Abstract

Advanced applications require construction, efficient access and management of large databases with rich data structures and inference mechanisms. However, such capabilities are not directly supported by the existing database systems. In this paper, we describe Relationlog, a persistent deductive database system that is able to directly support the storage, efficient access and inference of data with complex structures.

1 Introduction

Advanced applications require construction, efficient access and management of large databases with rich data structures and inference mechanisms. However, such capabilities are not directly supported by the existing database systems.

Deductive databases have the potential to meet the demands of advanced applications. They grew out of the integration of logic programming and relational database technologies. They are intended to combine the best of the two approaches, such as representational and operational uniformity, inference capabilities, recursion, declarative querying, efficient secondary storage access, etc.

However, most deductive databases systems such as Nail [16], LOLA [6], Glue-Nail [5], XSB [20], Aditi [21], LogicBase [7], Declare/SDS [8], etc. only support flat relations which are found inappropriate for advanced applications. A few deductive database systems that support data with complex structures such as LDL [3] and CORAL [18] are only implemented as memory-based systems and do not support direct access of data with complex structures [12].

For these reasons, we have developed a novel deductive database system called Relationlog at the University of Regina, Canada. Unlike existing deductive database systems, the Relationlog system is able to support the storage, efficient access and inference of data with complex structures. It provides declarative query language that is based on Relationlog [9, 13] and also a declarative data manipulation language that is based on DatalogU [11].

In this paper, we introduce the Relationlog system. In Section 2, 3, and 4, we introduce the Relationlog data definition language, query language, and data manipulation language respectively. In Section 5, We discuss Relationlog programs. In Section 6, we describe the Relationlog system architecture. Finally, in Section 7, we conclude and comment on our future plans.

2 Data Definition Language

The data definition language of Relationlog allows the specification of domain types, relation schemas, views and rules. It is similar to the SQL data definition language.

2.1 Domain Types

The primitive domain types supported are as follows:

- (1) **string**(*n*) for character strings with fixed-length *n*.
- (2) **string** which is an alias for **string**(16).
- (3) **token**(*n*) for alphanumeric strings without space, hyphen and quotation characters with maximum *n* characters.
- (4) **token** which is an alias for **token**(16).
- (5) **integer**(1) for 1-byte integers.
- (6) **integer**(2) for 2-byte integers.
- (7) **integer**(4) for 4-byte integers.
- (8) **integer**(8) for 8-byte integers.
- (9) **integer** which is an alias for **integer**(4).
- (10) **float**(4) for 4 byte real numbers.
- (11) **float**(8) for 8 byte real numbers.
- (12) **float** which is an alias for **float**(4).

The following are some examples of acceptable and unacceptable values of these primitive domain types:

Types	Acceptable	Unacceptable
string :	<i>'Bob Sam', "Bob's Wife"</i>	<i>'Bob", "abc"def"g", 'Bob's wife'</i>
token :	<i>BobSam, Bob_Sam</i>	<i>Bob Sam, bob-Sam, Bob's_Wife, 2sons, _Bob</i>
integer :	<i>5, 8, 120, -20</i>	<i>1.5, 1e4, one, "two"</i>
float :	<i>1, 2.0, 3.14, 1.2e4</i>	<i>one, 'two'</i>

In addition to the primitive domain types, the Relationlog system also supports *tuple*, *set*, *list*, and *bag* types that can be defined using the primitive domain types. The following are examples of these types.

```

Tuple types  [Last: string, First: string], [City: string, Country: string]
Set types    {token}, {[Last: string, First: string]}
List types   |integer|, |[Last: string, First: string] |
Bag types    *integer* *[Last: string, First: string]*

```

These non-primitive types must be created by the user using the **create domain** or **create type** command.

Figure 1 shows how to define new domain types in Relationlog.

Note that as a complex value based language, Relationlog does not allow circular type definition such as

```
create domain Person [Name : FullName, BirthDate : Date, Wife : Person]
```

```

create domain NameType string(10)
create domain FullName [
    Last : NameType,
    First : NameType]
create domain Date [
    Year : integer,
    Month : integer,
    Day : integer]
create domain Incomes |integer|
create domain Personlist |FullName|
create domain Location [
    City : string,
    Country : string]
create domain Paper [
    Author : NameType,
    Title : string(40),
    FirstPage : integer,
    LastPage : integer]

```

Figure 1: Sample Domain Definitions

User-created types can be deleted using the **drop domain** or **drop type** command. For example, the following command can be used to delete the user-defined type *FullName*:

```

drop domain FullName

```

However, if a user-defined type is used in the definitions of other types or relation schemas, the deletion of the type is not allowed.

The modification of types is not allowed in the Relationlog system as the type definitions are used to allocate storage space for relations.

2.2 Schema Definition

The Relationlog system supports two kinds of relations: *base relations* (or *extensional relations*) and *views* (or *intensional relations*). Base relations are stored persistently on disk. Views are defined using rules based on the base relations and other views. The schema for both kinds of relations must be defined explicitly using the **create relation** or **create table** command.

Figure 2 shows several relation schema definitions. As traditional relational database systems, Relationlog supports the definition of keys. However, keys must be defined on atomic attributes.

A newly created relation is empty initially. We have to use the **insert** command to be discussed in Section 4 to load data in the base relations or use rules to derive data in the views.

Figure 3 gives three relations *Persons*, *Conferences* and *Journals* which we will use as the running example. Note that the current implementation of Relationlog does not support the

```

create relation Persons
    (Name : NameType,
     BirthDate : Date,
     Parents : {NameType},
     LivesIn : Location)
     Earnings : Incomes
     key = (Name)

create relation Parentsof
    (Name : NameType,
     Parents : {NameType})
     key = (Name)

create relation Ancestorsof
    (Name : NameType,
     Ancs : {NameType})
     key = (Name)

create relation Conferences
    (CID : token,
     Year : integer,
     Location : Location,
     Papers : {Paper})
     key = (CID, Year)

create relation Journals
    (JID : token,
     Volume : integer,
     Number : integer,
     Papers : {Paper})
     key = (JID, Volume, Number)

create relation Publications
    (Author : Name,
     Papers : {[Title : string(40),
                  ID : token,
                  FirstPage : integer,
                  LastPage : integer]})
     key = (Name)

```

Figure 2: Sample Schema Definitions

storage of partial tuples (i.e, null-values) and partial sets introduced in [13]. Instead, sets in the relations must be complete.

To remove a relation from a Relationlog database, we can use the **drop relation** (or **drop table**) and **delete relation** (or **delete table**) commands. The former deletes all information about the dropped relation from the database while the latter deletes all tuples in the relation but retains the schema of the relation.

To modify the schema of a relation in a Relationlog database such as insert, delete or modify the attributes in a relation schema, we can use the **alter relation** or **alter table** command.

Conferences

CID	Year	Location		Papers			
		[City	Country]	Paper			
				[Author	Title	FirstPage	LastPage]
ICLP	91	[Paris	France]	{[Bob	Prolog	1	10]
				[Sam	Parlog	11	20]
				[Ann	Datalog	21	30]}
VLDB	97	[Athens	Greece]	{[Tom	Oracle	1	10]
				[Pam	Ingres	11	20]
				[Bob	Sybase	21	30]}
PODS	96	[Atlanta	USA]	{[Joe	DOOD	1	15]
				[Bob	NF2	16	30]}

Journals

JID	Volume	Number	Papers			
			Paper			
			[Author	Title	FirstPage	LastPage]
JLP	1	1	{[Tom	Logic	1	10]
			[Bob	Horn Clause	11	20]
TODS	1	1	{[Tom	Relation	1	10]
			[Pam	Calculus	11	20]
			[Jim	Algebra	21	30]}

Persons

Name	BirthDate			Parents	LivesIn		Earnings
	[Year	Month	Day]	{string}	[City	Country]	integer
Tom	[1913	11	20]	{}	[Toronto	Canada]	5000
Sam	[1983	5	30]	{Jim}	[Calgary	Canada]	1500
Bob	[1933	4	1]	{}	[Los Angeles	USA]	200, 2500
Pam	[1956	7	8]	{Bob}	[Chicago	USA]	900, 900
Joe	[1962	1	24]	{Bob}	[New York	USA]	3000

Figure 3: Sample Relations

Consider the following example:

alter relation *Persons* **add** *Spouse* : *NameType*

alter relation *Persons* **drop** *Spouse*

alter relation *Persons* **modify** *Parents*: *|NameType|*

The first command says add an attribute *Spouse* and its corresponding type into to the relation *Persons*. If the relation does not exist or the attribute has already been defined in the relation, this operation will fail. Besides, if the relation has tuples in it already, this operation will also fail as it will result in null-values in the existing tuples which is not allowed in the current implementation of Relationlog. Otherwise, the attribute is appended to the end of the existing attributes in the *Persons* relation. The second command says delete the attribute *Spouse* from the relation schema *Persons* and the corresponding attribute values from all the tuples. The last command says modify the type of the attribute *Parents* of the

relation *Persons* to the list type $|NameType|$. The operation will fail if the attribute does not exist or there are tuples in the relation which violate the new type constraint.

2.3 Views and Rules

Views are defined using rules based on base relations and other views. There can be two kinds of views in the Relationlog system: materialized (stored) and non-materialized. Materialized views are stored persistently on disk as base relations and are maintained current while non-materialized views are evaluated when they are queried.

Materialized views are created using the command **create stored view** while non-materialized views are created using the command **create view**. Unlike traditional relational languages such as SQL, Relationlog supports recursively defined views.

For instance, the following commands define three views: non-materialized views *Parentsof* and *Ancestorsof* and materialized view *Publications* based on the base relations *Persons*, *Conferences* and *Journals* defined earlier:

```
create view Parentsof as
    Parentsof(_Name, _Parents) :- Persons(_Name, _Age, _Parents, _Address)

create view Ancestorsof as
    Ancestorsof(_Name, <_Ancestor>) :- Parentsof(_Name, <_Ancestor>);
    Ancestorsof(_Name, <_Ancestor>) :- Parentsof(_Name, <_Parent>),
                                        Ancestorsof(_Parent, <_Ancestor>)

create stored view Publication as
    Publications(_Name, <[_Title, _ID, _FPage, _LPage]>) :-
        Conferences(_ID, _Loc, <[_Name, _Title, _FPage, _LPage]>);
    Publications(_Name, <[_Title, _ID, _FPage, _LPage]>) :-
        Journals(_ID, _Vol, _Num, <[_Name, _Title, _FPage, _LPage]>)
```

where *_Name*, *_Parents*, *_Age*, *_Address*, *_Parent*, *_Ancestor*, *_Title*, *_ID*, *_Loc*, *_FPage*, *_LPage*, *_Vol*, and *_Num* are logical variables. Especially, *_Parents* in the first command is a set-valued variable which ranges over the set of parents of a person tuple. The term *<_Ancestor>* in the second view definition is called a *partial set term* and is used for different purposes in different places. For the one in the head of the rules, it is used to group every ancestor of a specific person into a set. For the one in the body of the rules, it denotes an element of the set in the matching tuple. Similarly, the partial set term *<[_Title, _ID, _FPage, _LPage]>* in the head of the third view definition is used for grouping while the partial set term *<[_Name, _Title, _FPage, _LPage]>* in the body of the rules is used to denote a tuple in the set of the matching tuple. These commands will fail if the schemas for the relations are not defined or the rules in the view definitions are not well-typed with respect to their relation schemas. Note that the view *Ancestors* is recursively defined view.

Remark: We could have combined the schema and view definitions together into the view definition. However, as views/rules may be dependent on each other so that we may have views that have rules not well-typed. Besides, Relationlog does not following the Prolog convention which treats words starting with upper case letters as variables and words starting with lower case letters to as constants. Instead, all variables in Relationlog must start with *_* and *_* itself can be used as anonymous variable. In this way, words starting with upper case

letters can be treated as *Token* type constants.

Relationlog requires all rules in a database to be stratified. The stratification of rules is automatically checked every time a new view is created.

The materialized and non-materialized views can be dropped using the **drop view** command. However, if a view is used by other views, then the view can not be dropped.

For example, the following command will fail as the view *Parentsof* is used in the view *Ancestorsof*.

```
drop view Parentsof
```

If the view that can be dropped is non-materialized, then the rules used to define the view will be deleted. If the view that can be dropped is materialized, then both the rules that define the view and the derived relation will be deleted.

3 Query Language

The query language of Relationlog allows the user to directly query base relations and materialized or non-materialized views with the **query** commands. We use examples to illustrate.

Consider the following query in Relationlog:

```
query Persons(_Name, [_Year, -, ], _Parents, -, ), _Year < 1965
```

This query says list the name, birth year, and parents of every person who were born before 1965. In the current implementation of Relationlog, the attribute values in a tuple have a left-to-right order. We can omit the anonymous variable '_' in a tuple expression based on this order and abbreviate it into the following equivalent query:

```
query Persons(_Name, [_Year], _Parents), _Year < 1965
```

The results to this query based on the relation shown in Figure 3 will be displayed as a set of bindings as follows

```
_Name = Name : "Tom",   _Year = Year : 1913,   _Parents = Parents : {}
_Name = Name : "Bob",   _Year = Year : 1933,   _Parents = Parents : {"Jim"}
_Name = Name : "Pam",   _Year = Year : 1956,   _Parents = Parents : {"Bob"}
_Name = Name : "Joe",   _Year = Year : 1962,   _Parents = Parents : {"Pat"}
```

The following example shows how to find the two siblings who live in the same city.

```
query Persons(_Name1, -, _Parents, _Location),
Persons(_Name2, -, _Parents, Location), _Name1 <> _Name2
```

where *Location* is a tuple variable which ranges over a nested tuple in the matching tuple.

The views in Relationlog can be queried in the same way as base relations. The following are several examples:

```
query Ancestorsof(_Name, {"Bob"})
query Ancestorsof(_Name, {"Bob"}), not Parentsof(_Name, {"Bob"})
query Publications("Bob", {[_Title]}),
query Publications(_Name, {["Logic"]}), Persons(_Name, _Age, -, [_City, _Country])
```

The first query says find every descendant of *Bob*. The second query says find every descendant of *Bob* that is not a child of *Bob*. The third query says find the title of every paper that *Bob* wrote. The last query says find the name, age, and location of the author who wrote the paper *Logic*.

Note that if a view is materialized, then the query on it will be answered directly by retrieving data from the corresponding disk file. If it is not materialized, then the Relationlog system will first evaluate all relevant rules using proper strategies to find answers to the query. The results will be stored temporarily for later queries until the database is no longer active.

Displaying the results to a query as a set of bindings may not be the way the user likes. Besides, the results of the previous query may be useful for later queries. Therefore, Relationlog also allows a query to be represented as rule whose body specifies what to be queried and whose head specifies how to format the query results.

Consider the following two queries:

```
query Children(_Parent, <[_Child, _Year]>) :- Persons(_Child, [_Year], <[_Parent]>)
query Children2(_Parent, _Child, _Year) :- Children(_Parent, <[_Child, _Year]>)
```

The first query will display the results based on the relation shown in Figure 3 as a nested relation as follows:

```
(Jim,  {[Sam, 1983]})
(Bob,  {[Pam, 1956], [Joe, 1962]})
```

The second query will display the results of the first query as a flat relation:

```
(Jim,  Sam,  1983)
(Bob,  Pam,  1956)
(Bob,  Joe,  1962)
```

4 Data Manipulation Language

In the Relationlog system, the user can insert, delete and update base relations using data manipulation language commands **insert** and **delete**.

For example, the relation *Persons* shown in Figure 1 can be populated with the following commands:

```
insert Persons("Tom", [1913, 11, 20], {}, ["Toronto", "Canada"], |5000|)
insert Persons("Sam", [1983, 5, 30], {"Jim"}, ["Calgary", "Canada"], |1500|)
insert Persons("Bob", [1933, 4, 1], {}, ["Los Angeles", "USA"], |200, 2500|)
insert Persons("Pam", [1956, 7, 8], {"Bob"}, ["Chicago", "USA"], |900, 900|)
insert Persons("Joe", [1962, 1, 24], {"Jim"}, ["NewYork", "USA"], |3000|)
```

The tuple to be inserted will first be checked with respect to corresponding schema definition. Only well-typed tuples can be inserted successfully. Note that the attribute values in a tuple have a left-to-right order. Besides, null-value is not allowed at all in Relationlog.

The following examples show how to delete tuples from the database.

```
delete Persons("Tom", -, -, -)
delete Persons("Tom")
delete Persons(-, [-Year]), _Year < 1965
```

The first command says delete the tuple with the name *Tom* from the *Persons* relation. If the tuple does not exist, the operation will fail. This command can be abbreviated to the second one in which the anonymous arguments are omitted as well. The third command says delete every person who was born before 1965. Note that this command implies a query, i.e, find all persons who were born before 1965 and then delete them.

We now show how to update atomic values in tuples in Relationlog relations. Consider the following examples:

```
delete Persons(-, -, ["Toronto"]), insert Persons(-, -, ["Vancouver"])
query Persons("Sam", -, (-SamsParent)), delete Person(-SamsParent, ),
insert Person(-SamsParent, -, ["Vancouver", "Canada"])
```

The first command says transfer every person in Toronto to Vancouver. The second command says transfer *Sam*'s parents' to Vancouver, Canada. Note that there is an explicit query command in the second update command. Indeed, complex updates can be performed in Relationlog by using complex queries in the update commands.

The following examples show how to update elements in sets:

```
insert Persons("Pam", -, ("Bob"))
delete Persons("Joe", -, ("Pat"))
```

The first command says insert *Bob* into *Pam*'s parents set. The operation will fail if the tuple for *Pam* is not in the relation or *Bob* is already a parent of *Pam*. The second command says delete *Pat* from *Joe*'s parents set rather than the whole tuple. This operation will fail if *Pat* is not in *Joe*'s parents set.

An update command in Relationlog is treated as transaction which can either succeed or fail. If it fails, it has no effect on the database at all. Updates in the Relationlog system have a declarative semantics which is a straightforward extension of the semantics presented in [11].

Updating base relations may result in materialized views and results of previous queries that are kept invalid. Relationlog will delete the results of previous queries that are invalid and update materialized views.

5 Relationlog Programs

In order to simplify the creation of database relations, the Relationlog system allows the user to put all the information necessary for the database creation into a Relationlog program, which consists of four parts: types, schema, facts and rules. The types part contains the type definitions. The schema part contains the relation schemas for both extensional and intensional relations. The facts part contains tuples in base relations. The rules part contains rules used to define views.

Types

```
NameType = string(10)
FullName = [Last : NameType, First : NameType]
Date = [Year : integer, Month : integer, Day : integer]
Incomes = |integer|
```

Schema

```
Person(Name : NameType,
        BirthDate : Date,
        Parents : {string},
        LivesIn : Location,
        Earnings : Incomes)
    key = (Name)

Parentsof(Name : NameType,
           Parents : {NameType})
    key = (Name)

Ancestorsof(Name : NameType,
             Ancs : {NameType})
    key = (Name)
```

Facts

```
Persons("Tom", 85, {}, ["Toronto", "Canada"])
Persons("Sam", 15, {"Jim"}, ["Calgary", "Canada"])
Persons("Bob", 65, {}, ["LosAngeles", "USA"])
Persons("Pam", 42, {"Bob"}, ["Chicago", "USA"])
Persons("Joe", 36, {"Bob"}, ["NewYork", "USA"])
```

Rules

```
Parentsof(_Name, _Parents) :- Persons(_Name, _Age, _Parents, _Address)
Ancestorsof(_Name, _Ancestor) :- Parentsof(_Name, _Ancestor)
Ancestorsof(_Name, _Ancestor) :- Parentsof(_Name, _Parent),
                                   Ancestorsof(_Parent, _Ancestor)
```

Figure 4: Sample Relationlog Program

For example, Figure 4 shows a Relationlog program which is part of the sample database shown in Section 2.

6 System Architecture

In this section, we briefly describe the system architecture of the current Relationlog system. The details are available in [14].

The Relationlog system has been implemented as a single-user persistent deductive database system. The system architecture of Relationlog is shown in Figure 2.

The system is organized into three layers. The first layer is the user interface which receives and processes user commands and displays results. The commands available from the user interface are described in details in [15].

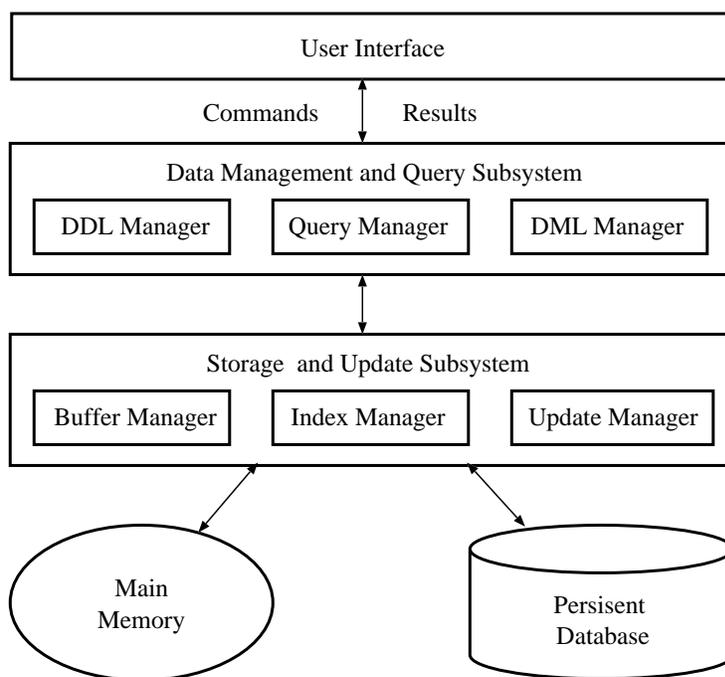


Figure 5: Relationlog System Architecture

The second layer is the Data Management and Query Subsystem which consists of three managers: DDL Manager, Query Manager, and DML Manager. They cooperate with each other tightly and direct the Storage and Update Subsystem to handle smaller tasks respectively. The DDL Manager is responsible for processing all DDL commands, maintaining system catalogs about domains, relations, indices and rules, and answering all the type checking requests from the DML and Query Managers. It checks if all rules are stratified when a new view is created. As views may be materialized, the DDL manager is also responsible for dropping those materialized intensional relations and the corresponding rules when a view is dropped.

The Query Manager is responsible for data retrieval and rule evaluation. It supports three strategies: matching, seminaive evaluation, and top-down with tabling. If the data to be retrieved are in an extensional relation or in a materialized view, then it simply uses matching and indices to find the results. If the data to be retrieved are in a non-materialized view, it automatically decides whether to use seminaive bottom-up fixpoint evaluation with rule ordering mechanism as proposed in [17] or set-at-a-time top-down with tabling to find the results based on the information about the rules.

The DML Manager performs all the updates to the extensional relations. As an update may imply a query, the DML manager may request the Query Manager to process the query before performing the update. After an extensional relation is updated, it will also request the Query Manager to propagate the updates to materialized views that are dependent on the updated relation if any.

The third layer is the Storage and Update Subsystem which consists of three main managers: Buffer Manager, Index manager, and Update Manager. The Buffer Manager deals with loading, dropping, and updating domains, the relation schemas, relation tuples, indices and rules between main memory and disk files. The Index Manager is in charge of the creation

and updates of B-tree and Hash indices for relations and provides a transparent interface. The Update Manager deals with the updates of domains, relation schemas, relation tuples, indices and rules.

7 Conclusion

In this paper, we have given a brief introduction to the Relationlog system. A complete implementation as described in this paper has been developed. The system will be available over the Internet soon after further testing and debugging. For more information about the Relationlog system, see the web home page: <http://www.cs.uregina.ca/~mliu/RLOG>.

We are currently exploring other query processing strategies for the Relationlog system and extending it into a full-fledged system. We are also developing SQL, OQL [2], extended relational algebra and Calculus [1, 4, 19] interfaces on top of it in order to make it a useful tool for teaching both introductory and advanced database courses.

The current Relationlog prototype does not support null-values. We intend to extend the Relationlog system to support null-values as described in [10]. Besides, we intend to add aggregate operations to it as well. Several application systems based on Relationlog are under development, including an auto-CAD system.

Acknowledgments

The authors would acknowledge the contributions of the following people to the Relationlog system: Tao Guan, Shilpesh Katragadda, Ming Zhao, Mingyuan Deng, and Sudha Srinivas. This work has been partly supported by the Natural Sciences and Engineering Research Council of Canada.

References

- [1] S. Abiteboul and N. Bidoit. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *J. Computer and System Sciences*, 33(3):361–393, 1986.
- [2] R.G.G. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann, 1996.
- [3] 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.
- [4] L. Colby. A Recursive Algebra and Query Optimization for Nested Relations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 124–138, Portland, Oregon, 1989.
- [5] 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.
- [6] 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.
- [7] 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.

- [8] 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.
- [9] 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.
- [10] M. Liu. A Logical Semantics for Complex Object Databases with Partial and Complete Information about Sets and Tuples. In *Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming (DDL'98)*, Manchester, UK, June 20 1998.
- [11] M. Liu. Extending Datalog with Declarative Updates. *Submitted for Publication*, 1998.
- [12] M. Liu. Overview of Datalog Extensions with Sets and Tuples. In *Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming (DDL'98)*, Manchester, UK, June 20 1998.
- [13] M. Liu. Relationlog: A Typed Extension to Datalog with Sets and Tuples. *Journal of Logic Programming (in press)*, 36(3):271–299, 1998.
- [14] M. Liu and R. Shan. The Design and Implementation of the Relationlog Deductive Database System. In *Proceedings of the 9th International Workshop on Database and Expert System Applications (DEXA Workshop '98)*, Vienna, Austria, August 24-28 1998. IEEE-CS Press.
- [15] M. Liu and R. Shan. The Relationlog System User Manual, Release 1.0. June 1998.
- [16] 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.
- [17] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Program. In *Proceedings of the International Conference on Very Large Data Bases*, pages 359–371, Brisbane, Queensland, Australia, 1990. Morgan Kaufmann Publishers, Inc.
- [18] Raghuram Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The CORAL Deductive System. *VLDB Journal*, 3(2):161–210, 1994.
- [19] M. A. Roth, H. F. Korth, and A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM TODS*, 13(4):389–417, 1988.
- [20] 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.
- [21] 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.