SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:409–443 (DOI: 10.1002/spe.378)

The Relationlog system prototype

Mengchi Liu*,†

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

SUMMARY

The Relationlog system is a novel persistent deductive database system for advanced data and knowledge-based applications. It directly supports the storage and inference of data with complex structures, especially data supported in nested relational and complex-object models. The Relationlog system supports the Relationlog query language, which is a typed extension of Datalog with tuples and sets and stands in the same relationship to the nested relational and complex-object models as Datalog stands to the relational model. It also supports an SQL-like data definition language and a declarative data manipulation language. This article introduces the Relationlog language, discusses the system architecture, the design decisions incorporated within its implementation, and our experience in developing the system. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: deductive databases; nested relations; complex-object relations; optimization; storage management; rule management

1. INTRODUCTION

Advanced database applications require the capabilities to effective storage, efficient access and inference of large amounts of data with complex structures. However, such capabilities are not directly supported by the existing database systems.

In the past decade, various advanced data models such as nested relational and complex-object models [1–8] and object-oriented models [9–13] have been developed to support data with complex structures.

On the other hand, a lot of interest has arisen in the deductive database approach which integrates the logic programming and relational database techniques to support the inference of large amounts

*Correspondence to: Mengchi Liu, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada S4S 0A2.
†E-mail: mliu@cs.uregina.ca

Contract/grant sponsor: Natural Sciences and Engineering Research Council, Canada; contract/grant number: OGP0193553-1996

Received 25 June 1999
Revised 20 November 2000
Accepted 20 November 2000

of data. Datalog (with negation), a restricted form of the logic programming language Prolog without function symbols, has been widely accepted as the standard deductive database query language [14,15]. Various set-oriented evaluation strategies specific for such kinds of deductive databases have been the main focus of extensive research [14–23] and a number of deductive database systems or prototypes based on Datalog have been reported. These include Nail [24], LOLA [25], Glue-Nail [26], XSB [27], Aditi [28], LogicBase [29], Declare/SDS [30] etc. Some of them perform well for large applications. See Reference [31] for a survey of these deductive database systems.

Deductive database systems based on Datalog only support flat relations which are inappropriate for advanced data and knowledge-based applications. For this reason, Datalog has been extended into more powerful languages to support data with complex structures, such as LDL [32], LPS [33], COL [34], Hilog [35], and Relationlog [36]. See Reference [37] for an overview of some of these languages. Several deductive database systems that support data with complex structure have been developed, such as LDL [38], COL [34], and CORAL [39]. However, they are only implemented as memory-based systems and fail to address the issues of effective storage, efficient access and inference of large amounts of data with complex structures, although LDL and CORAL both have interfaces to persistent database systems so that persistent data can be loaded into the memory for processing.

The objective of the Relationlog system is to develop techniques for new generation database systems that directly support the storage and inference of large amounts of data with complex structures based on the Relationlog query language [36]. The Relationlog query language is a typed extension of Datalog with tuples and sets. It stands in the same relationship to the nested relational and complex-object models as Datalog stands to the relational model. Furthermore, it has a well-defined declarative semantics.

The Relationlog project started at the beginning of 1996. There have been three different implementations of Relationlog. The first one was built on top of the commercial DBMS Oracle and Ingres by using C with embedded SQL statements. This implementation successfully turned Oracle and Ingres into a persistent deductive database system that supported large amounts of data with complex values and the Relationlog language. However, the performance was not satisfactory for the following reasons. First, nested and complex-object relations in Relationlog had to be stored as flat relations so that there was a significant overhead to convert between flat relations and complex object relations. Second, not much query optimization could be performed by the Relationlog program as it was a fat client with no access to the physical level of the DBMS server. Last, even though relational database operations were set-at-a-time, access to the relations in our C program with embedded SQL statements could only be cursor-based and was only tuple-at-a-time.

The second implementation was built on top of the EXODUS persistent storage manager [9]. EXODUS was a very good choice for Relationlog. It was client/server based and supported the storage of data with complex structures. However, it was no longer supported and its host language, E, did not provide a reasonable debugging environment so that it turned out to be very difficult to develop a large system like Relationlog. After struggling with EXODUS for several months with a small portion of Relationlog working, we had to abandon this implementation.

Finally, we have chosen PARODY (Persistent Almost-Relational Object Database Manager) [40] for the Relationlog system development. Unlike Oracle, Ingres and EXODUS, PARODY is only a single-user database system that does not support concurrency control, transaction, and recovery management. However, PARODY supports persistent objects with B-tree and Hash indexes and has concise fully-implemented C++ codes that are portable to multiple platforms. These features are essential for the

objective of the Relationlog prototype, as we do not have to start from scratch. To make PARODY more suitable for Relationlog, the original code has been modified extensively to support Relationlog data and metadata storage and management.

The Relationlog system has been developed mainly on a SUN SPARCstation running Solaris 2.5. It contains 20 000 lines of C++ code. It has been successfully ported to SGI, Linux, DEC and Windows NT/95. It is implemented as a single-user persistent database system that supports the Relationlog query language, an SQL-like data definition language and a declarative data manipulation language based on DatalogU [41].

The following are the novel features in the Relationlog system:

1. standard user interface that appears easy to use
2. the user can specify how to organize query results using a rule
3. schemas for both extensional and intensional relations
4. powerful mechanisms for direct inference and access to embedded values in nested and complex-object relations as if they are normalized
5. persistent domain types, schemas, facts, and rules
6. intensional data can be materialized and maintained currently automatically
7. facts and rules needed for query evaluation are dynamically selected and results are temporarily stored for later queries
8. a least recently used (LRU) mechanism is used to remove intensional and extensional data from memory when memory space is needed
9. various evaluation strategies are used automatically and rules used for query evaluation are dynamically rewritten based on the nature of the query and the data in the database without the user's intervention, so that there is no need for specifying query forms or modes as in Aditi, LDL, and CORAL.

This paper is a major revision and extension of the papers in [42–44]. It is organized as follows. Section 2 provides an overview of the Relationlog language, introducing the data definition language, query language, data manipulation language, and Relationlog programs. Section 3 describes the implementation of the system, its system architecture, file structure, storage management, and query processing. Section 4 briefly discusses the major differences and similarities between Relationlog and other related deductive database systems. Conclusions and future plans are discussed in Section 5.

2. THE RELATIONLOG LANGUAGE

Unlike Datalog which is an untyped deductive language for flat relations, Relationlog is a typed deductive language for nested and complex object relations.

A Relationlog database consists of a collection of named relations, whose tuple components can be tuples, sets, lists, and bags besides atomic values. The relations are partitioned into two kinds: base relations (or extensional relations) which are stored persistently on disk and can be directly updated by the user, and views (or intensional relations) which are defined by rules and cannot be directly updated by the user.

The Relationlog language consists of three sub-languages: data definition language, data manipulation language, and query language. We describe them in this section.


```
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.

2.1. 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.

2.1.1. Domain types

Relationlog supports primitive domain types such as **string**, **integer**, **token**, and **float** with various lengths.

In addition to the primitive domain types, Relationlog 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**]*

New domain types can be defined in Relationlog with the **create domain** or **create type** command of the form:

create domain *Domain-Name Type*
create type *Domain-Name Type*

where *Domain-Name* is the name for the newly created domain type *Type*.

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

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

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

However, in the relation schema, such a circular reference can be indirectly supported using **references** (that is, foreign key). See the next section.

User-defined 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

If a user-defined type is used in the definitions of other types or relation schemas, then its deletion 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.

2.1.2. Schema definition

The schema for a relation, either a base relation or view, must be defined explicitly using the **create relation** or **create table** command of the following form:

create relation *Relation-Name*
(A_1 : D_1 [[**element**|C_1] **references** [*Relation-Name*$_1$] B_1],
...
A_n : D_n [[**element**|C_n] **references** [*Relation-Name*$_n$] B_n])
[**key** = ($A_{j_1}, \ldots, A_{j_m}$)]

where *Relation-Name* is the name of the relation to be created, each A_i is an attribute name and each D_i is a domain type, A_i itself—its element if D_i is a set type, or its component C_i if D_i is a tuple type—may reference an attribute B_i in the same relation or in relation *Relation-Name*$_i$, and the atomic attributes $A_{j_1}, \ldots, A_{j_m}$ form the primary key for the relation. When the primary key is not defined, Relationlog will treat all atomic attributes as the primary key. Note that relations in Relationlog are required to be in Partition Normal Form (PNF) [8].

Figure 2 shows several relation schema definitions. Like 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 Data Manipulation Language command **insert** to load data into the base relations or use rules to derive data in the views (see Section 2.1.5).

Table I gives three base relations, Persons, Conferences, and Journals, which are used as the running example.

Relationlog is suitable for more complex relations with nested tuples and sets.

Consider the ROBOT relation schema in Figure 3. This relation is used to represent the parts of real robots. It has three main attributes: ID, which is a string, ARMS, which is a nested relation, and GRIPPERS, which is also a nested relation. The nested relation GRIPPERS has two attributes of **string** type: ID and FUNCTION. The nested relation ARMS has two attributes: ID, which is a

```
create relation Persons
        (Name: NameType,
         BirthDate: Date,
         Parents: {NameType} element references Name,
         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),
                   FirstPage: integer,
                   LastPage: integer]})
         key = (Name)
```

Figure 2. Sample schema definitions.

string, and AXES, which is a nested relation. The nested relation AXES has two tuple type attributes: KINEMATICS and DYNAMICS. A subset of the ROBOT relation is shown in Table II.

The index mechanism is tightly integrated with the creation and deletion of relations. When a relation is created, Relationlog automatically creates a B-Tree index on the primary key attributes.

The user can create/drop other indexes using the command of the following form:

create index *Index-Name* **on** *Relation-Name* $(A_1, \ldots, A_m)$
drop index *Index-Name*

where *Index-Name* is a system unique name for the index, *Relation-Name* is the name of a created relation, and $A_1, \ldots, A_m$ are atomic attributes of *Relation-Name*.

Table I. Sample relations.

Persons

Name	BirthDate			Parents	LivesIn		Earnings
	[Year	Month	Day]	{**string**}	[City	Country]	|**integer**|
Tom	[1953	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|

Conferences

CID	Year	Location		Papers			
				{ Paper			}
		[City	Country]	[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]}

```
create relation ROBOT (
    ID: string(20),
    ARMS: {[
        ID: string(20),
        AXES: {[
            KINEMATICS: [
                DH_MATRIX: {[
                    COLUMN: integer,
                    VECTOR: {float}
                    ]}
                JOINT_ANGLE: [
                    MAX: integer,
                    MIN: integer
                    ]
                ]
            DYNAMICS: [
                MASS: float,
                ACCEL: float
                ]
            ]}
        ]}
    GRIPPERS: {[
        ID: string(20),
        FUNCTION: string(20)
        ]}
    )
```

Figure 3. ROBOT relation schema.

Table II. ROBOT relation.

ROBOTS

ID	ARMS							GRIPPERS	
	ID	AXES						ID	FUNCTION
		KINEMATICS				DYNAMICS			
		DH_MATRIX		JOINT_ANGLE		MASS	ACCEL		
		COLUMN	VECTOR	MAX	MIN				
Artoo	Left	1	{1,0,0,0}	−180	180	50.0	1.0	#100	gripper
		2	{0,1,0,0}					#200	righthand
	Right	3	{0,1,0,0}	−90	90	10.0	1.3		
		4	{0,0,0,1}						
Detoo	Middle	6	{1,1,1,0}	−270	270	10.0	2.7	#300	lefthand
		8	{0,1,1,1}						
		9	{0,1,0,1}						

To remove a relation from a Relationlog database, the **drop relation** and **delete relation** commands of the following forms can be used:

drop relation *Relation-Name*
delete relation *Relation-Name*

where *Relation-Name* is the name of an existing relation. The former command deletes all information about the dropped relation from the database, including the indexes related to a relation. The latter deletes all tuples in the relation but retains the schema and indexes 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, the **alter relation** command of the following form can be used:

alter relation *Relation-Name* **add** *Attribute* : *Type*
alter relation *Relation-Name* **drop** *Attribute*
alter relation *Relation-Name* **modify** *Attribute* : *Type*

where *Relation-Name* is the name of an existing relation and *Attribute* is an attribute in *Relation-Name* and *Type* is a type.

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 the relation Persons. If the relation does not exist or the attribute has already been defined in the relation, this operation fails. Also, if the relation has tuples in it already, this operation also fails as it results 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 fails if the attribute does not exist or there are tuples in the relation which violate the new type constraint.

2.1.3. Views and rules

Views in Relationlog are defined using rules based on base relations and other views.

A rule in Relationlog has the following form:

$$A :- L_1, \ldots, L_n$$

where the head A is an atom and each L_i in the body is a literal, which is either an atom, a negated atom using the keyword **not**, or an arithmetic or set-theoretic comparison expressions over variables and constants.

An atom has the following form:

$$relation\_name(E_1, \ldots, E_m),$$

where *relation_name* is the name of a relation, $E_1, \ldots, E_m$ are the terms, which are either constants, variables, partial set terms of the form $\langle O_1, \ldots, O_n \rangle$, complete set terms of the form $\{O_1, \ldots, O_n\}$, and tuple terms of form $[O_1, \ldots, O_n]$ or $[A_1 : O_1, \ldots, A_n : O_n]$ with $O_1, \ldots, O_n$ and $A_1, \ldots, A_n$ being terms and attribute names, respectively.

Relationlog supports two kinds of views: materialized 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** of the forms:

create stored view *View-Name* **as** $R_1; \ldots ; R_m$
create view *View-Name* **as** $R_1; \ldots ; R_m$

where *View-Name* is the name of the view and $R_1, \ldots, R_m$ are rules with $m > 0$.

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 Publications 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 domain 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 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 Ancestorsof is a 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, views that have rules may not be well-typed. Besides, Relationlog does not follow the Prolog convention which treats words starting with upper case letters as variables and words starting with lower case letters as constants. Instead, all variables in Relationlog must start with _, and _ itself can be used as an 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 of the following form:

drop view *View-Name*

where *View-Name* is the name of an existing view.

For example, the following command can be used to drop the view Publications:

drop view Publications

2.1.4. Query language

The query language of Relationlog allows the user to directly query base relations and materialized or non-materialized views with the query command of the form:

query $L_1, \ldots, L_n$

where each L_i is a literal. The result to a query is always based on the final evaluated and combined information and displayed as a set of bindings.

Consider the following query in Relationlog:

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

This query says list the name, birth year, and parents of all persons who were born before 1965. In the current implementation of Relationlog, the attribute values in a tuple have a left-to-right order. Thus 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 Table I are displayed as follows:

_Name= Name:Tom, _Year= Year:1953, _Parents= Parents:{}
_Name= Name:Bob, _Year= Year:1933, _Parents= Parents:{}

_Name= Name:Pam, _Year= Year:1956, _Parents= Parents:{Bob}
_Name= Name:Joe, _Year= Year:1962, _Parents= Parents:{Bob}

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 $\neq$ _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 is answered directly by retrieving data from the corresponding disk file. If it is not materialized, then the Relationlog system first evaluates all relevant rules using proper strategies to find answers to the query. The results are 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. Also, the results of the previous query may be useful for later queries. Therefore, Relationlog also allows a query to be represented as a rule whose body specifies what is to be queried and whose head specifies how to format the query results. Such a query has the following form:

query H :– $L_1, \ldots, L_n$

where H indicates the result relation, the attributes of which could be implicitly derived all from the rule body or explicitly declared by the user. In the first case, the head is just used to handle the outputs. As for the second case, the head relation must be declared before the issuance of the query, and then the results of the query are stored temporally in the database until the database is closed.

Consider the following two queries:

query Children (Name: _Parent, Children: ⟨[_Child, _Year]⟩) :–
Persons (_Child, [_Year], ⟨[_Parent]⟩)

query Children2 (Name: _Parent, Child: _Child, ChildBirthYear: _Year) :–
Children (_Parent, ⟨[_Child, _Year]⟩)

The first query displays the results based on the relation shown in Table I as a nested relation as follows:

Children

Name	Children
Jim	{[Sam, 1983]}
Bob	{[Pam, 1956], [Joe, 1962]}

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

Children2

Name	Child	ChildBirthYear
Jim	Sam	1983
Bob	Pam	1956
Bob	Joe	1962

2.1.5. Data manipulation language

In the Relationlog system, the user can insert, delete, and update base relations using data manipulation language commands **insert** and **delete** of the following forms:

insert A
delete $A, L_1, \ldots, L_n$

where A is an atom and each L_i is a literal for $1 \leq i \leq n$.

For example, the relation Persons shown in Table I can be populated with the following commands:

insert Persons ("Tom", [1953, 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"}, ["New York", "USA"], |3000|)

The tuple to be inserted is first checked with respect to the 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. Also, null-values are 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 fails. 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 can express complex updates using query and update commands together as follows:

query $L_1, \ldots, L_n$, **delete** A_D, **insert** A_I

where $L_1, \ldots, L_n$ are literals with $n \geq 0$, A_D and A_I are atoms. Also, the deletion part can be omitted.

Consider the following examples:

delete Persons (_N,_B,_P, ["Toronto"]), **insert** Persons (_N,_B, _P, ["Vancouver"])

query Persons ("Sam", _, ⟨ _SamsParent⟩), **delete** Persons (_SamsParent,_B, _P),
insert Persons (_SamsParent,_B, _P, ["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" ⟩)

query Parentsof ("Joe", ⟨_X⟩), not Parentsof("Pam",⟨_X⟩,
insert Parentsof ("Jim",⟨_X⟩)

The first command says insert Bob into Pam's parents set. The operation fails 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 fails if Pat is not in Joe's parents set. The last command says make each parent of Joe, who is not a parent of Pam, one of parents of Jim. An update command in Relationlog is treated as a 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 [41].

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

2.1.6. 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.

For example, Figure 4 shows a Relationlog program which is part of the sample database shown in Section 2.1.


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

Schema
        Persons  (Name: NameType,
                  BirthDate: Date,
                  Parents: {NameType} element references Name,
                  LivesIn: Location,
                  Earnings: Incomes)
                  key = (Name)
        Parentsof (Name: NameType,
                   Parents: {NameType})
                   key = (Name)
        Ancestorsof  (Name: NameType,
                      Ancs: {NameType})
                      key = (Name)

Facts
        Persons ("Tom", [1953, 11, 20], {}, ["Toronto", "Canada"])
        Persons ("Sam", [1983, 5, 30], {"Jim"}, ["Calgary", "Canada"])
        Persons ("Bob", [1933, 4, 1], {}, ["Los Angeles", "USA"])
        Persons ("Pam", [1956, 7, 8], {"Bob"}, ["Chicago", "USA"])
        Persons ("Joe", [1962, 1, 24], {"Bob"}, ["New York", "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. Relationlog program family.

3. IMPLEMENTATION OF RELATIONLOG

The Relationlog system is inspired by LDL [38]. It also adopts many of the applicable and suitable technologies from CORAL [39], Aditi [28], and Glue-Nail [45]. It has been implemented as a single-user persistent deductive database system in C++ under the Unix environment.

The system architecture of Relationlog is showed in Figure 5. It is organized into three layers. The first layer is the user interface. Two kinds of user interfaces are provided: textual user interface and graphical user interface. They provide different kinds of environment for the user to define, query, and manipulate databases. The textual user interface accepts user commands, performs syntactical analysis of the commands, passes valid commands to the Data Management and Query Subsystem, and displays results or error messages back to the user. The graphical user interface allows users to pick operations

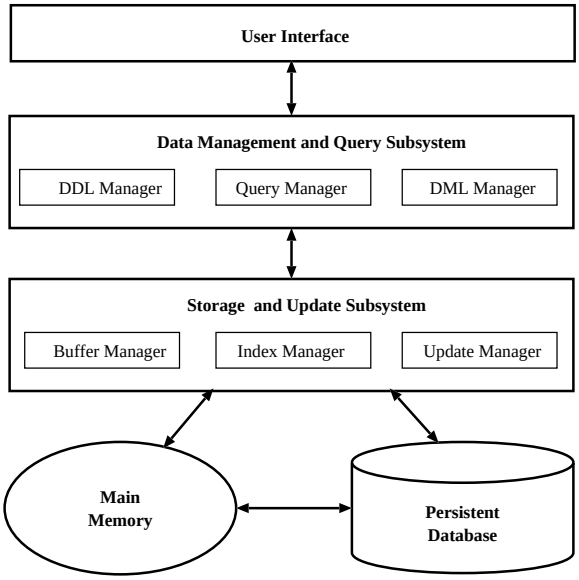


Figure 5. Relationlog system architecture.

off menus. Both kinds of interfaces actually send operations in internal representation to the lower layer.

The second layer is 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 each to handle smaller tasks. We describe the details in Section 3.3.

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, relation schemas, relation tuples, indexes and rules between main memory and the persistent database that consists of disk files. The Index Manager is in charge of the creation and updates of the B-tree and Hash indexes for relations and provides a transparent interface. The Update Manager deals with the updates of domains, relation schemas, relation tuples, indexes, and rules. We describe the details in Section 3.2.

3.1. File structure of persistent database

The Relationlog persistent database consists of UNIX files with similar structures. The files store facts, indexes, and catalogs. Relationlog maintains five system catalog files for meta information management: *database.sys* that records the information of all created databases in the system, *relation.sys* that records the information of all relations in each database, *attribute.sys* that records

the information of all attributes of each relation, *rule.sys* that records the information of all rules of each database, and *ruleunit.sys* that records the information of all rule literals existing in each rule.

Each database created in Relationlog has four files of its own in the UNIX file system: a data file with extension .EDB which contains all facts of extensional relations, a data file with extension .IDB which contains all facts of materialized intensional relations, an index file with extension .EDX which contains key indexes to the facts of extensional relations, and an index file with extension .IDX which contains key indexes to the facts of intensional relations. The Relationlog file architecture is based on PARODY [40]. The original structure has been modified for meta information management and set handling.

All the files in the Relationlog persistent database are composed of nodes. Nodes are addressed logically by node numbers starting from 0. Relationlog uses the node system in two ways: the index files and the system catalog files are organized into fixed-length nodes; the data files are unformatted, variable-length data threads that could extend beyond the boundaries of a node. There is a deleted-node queue in each file so that the deleted nodes can be re-allocated.

We now discuss how facts, indexes, and catalogs are stored in Relationlog.

3.1.1. Nodes

The Relationlog persistent database is composed of special files. A file in Relationlog contains a header record followed by fixed-length node records in a thread, whose internal structure is shown in Figure 6.

Such a file structure can support:

1. variable-length persistent relations
2. indexes into the facts on nested relations
3. persistent objects that can grow or shrink in length or be deleted
4. reuse of deleted file space
5. meta information management.

Header record. The header record contains two node numbers. The first node number points to the highest node that has been allocated. For instance, the node $n - 1$ is the highest one in Figure 6. The second points to the first node in a thread of deleted nodes.

Node records. Each node record corresponds to a unique node number of its own. It is a fixed-length disk record with a next-node number at the front of the record followed by anything Relationlog wants to put into the node. The next-node number is a thread pointer. It points to the next node in a logical node thread. Relationlog uses the node number of each node record to access the node by translating the number into the physical location of the node in the file.

Node threads. Since objects may have different lengths, a node sometimes is not enough. A node thread consisting of a number of nodes is thus used to contain the whole object. Relationlog maintains a pointer to the first node in the thread. The node pointer at the front of the first node points to the second node, which points to the third, and so on, until a node is reached that has zero in its next node pointer, which marks the end of the thread. The nodes in a thread are in no particular sequence in the file.

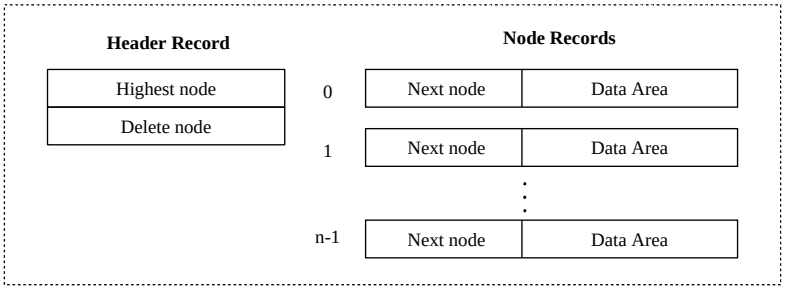


Figure 6. File structure.

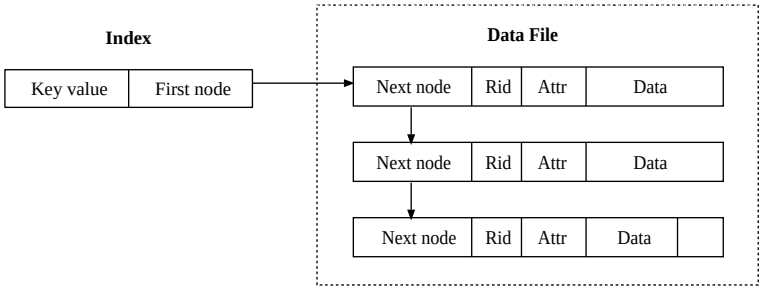


Figure 7. Index and data file.

Node deletion. When a node is deleted, Relationlog first clears the data area of the node and then adds it to the deleted node thread, which is pointed to by the deleted node pointer in the header record. The pointer actually points to the most recently deleted node. When a node is deleted, Relationlog moves the deleted node pointer from the header into its next node pointer and moves its own address into deleted node pointer in the header.

Node allocation. When a new node is needed, Relationlog looks first at the deleted node pointer in the header. If the pointer is non-zero, Relationlog allocates that node for the new usage and changes the next node pointer from the node into the deleted node pointer. If the deleted node thread has no nodes, Relationlog allocates a node from the end of the file by using the node number of the highest node allocated as recorded in the file header.

3.1.2. Data file

The Relationlog system stores facts of a database in the node threads of a data file of the database. The structure of a data file is shown in Figure 7. Each fact starts from a node, whose node number

is the fact's address. The fact starts with three integers followed by the data of the fact. The first two integers are the next node number and the relation identification of the fact, used in locating the fact and rebuilding the index file, which is described in the following section. The third integer indicates the attribute node related to this fact so that the nested set elements can be accessed. Thus this extended structure is suitable for storing nested relations.

If the length of a fact is greater than that of a node, it spills over into the next node in the thread, continuing that way until the entire fact is recorded. The balance of the last node in a fact's thread is padded with zeros (null). The same physical file of nodes holds all the objects of all the relations in a database. Nothing that stands alone in the file itself tells the database the types of facts or anything about their formats. For Relationlog to correctly locate a fact, it needs the address of the fact's first node. That address is maintained by the indexes in a separate file which is described below.

In the Relationlog system, tuples in the same relation with n top-level non-empty sets, lists, and bags are stored in $n + 1$ node streams. One stream is for the tuple itself and one stream for each set, list, or bag in the tuple with its first node number, the description, and the count of the elements stored in the set stream so that we can access its elements from the tuple stream. The number of nodes needed for the tuple stream is the same for all tuples of the relation, but the number of nodes needed for other streams depends on the elements they hold. For example, an empty set does not occupy any node. The maximum number of nodes allowed currently in a Relationlog file is 2^{32}. The current Relationlog system does not support raw data. If the set, list, or bag in the tuple has nested sets, lists, or bags, then additional node streams are used to store them.

For the nested tuples, there is no separate node stream because the length of the tuples is known from its type definition. Instead, the nested tuples are stored in their root tuple stream.

Consider the following new fact in the Persons relation in Table I:

(Ann, [1976, 11, 25], {Tom, Pam}, [Regina, Canada], |1000|)

Because of the nested set, this fact is stored in two node streams: one stream for the atomic values and the nested tuple and the other stream for the nested set. Suppose that the available nodes are 10, 11, 94, ... and the first stream takes two nodes 10 and 11, and the second node stream takes just the node 94, then the node 10 contains the following data: the next node number 11; relation identification information for the relation Persons; value Ann for the attribute Name; values 1976, 11, 25 for the attribute BirthDay; the node number for the Parents set. The node 11 contains the values for attributes LivesIn and Earnings with the rest bytes filled with 0 (null). The node 94 contains the information about the nested set: its cardinality 2, its parent node number 10, and its elements Tom and Pam with the unused space filled with zeros. The internal representation of this fact is shown in Figure 8.

The Relationlog file structure in Figure 6 is suitable for updates. Continue with the above example in Figure 8. Suppose the attribute Parents is altered to Relatives in the Persons relation and two brothers Sam and Joe are added to the same set. Assume the node 94 cannot contain four elements of the set. Thus a new node needs to be allocated and its number is put into the next node field of the first node containing the set elements. Assume that the next available node is 95. Then the first three elements are placed in node 94 and the last one is put in node 95 and the unused bytes are filled with 0 (null). Also the set cardinality is also changed from 2 to 4. Figure 9 shows how this updated fact is stored in the Relationlog database.

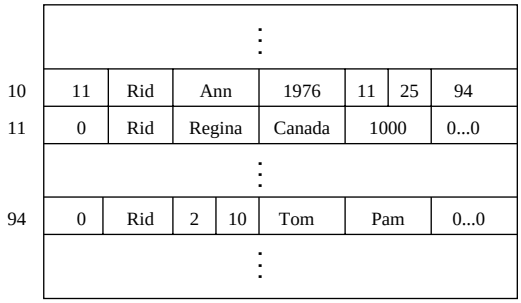


Figure 8. Internal representation of a fact.

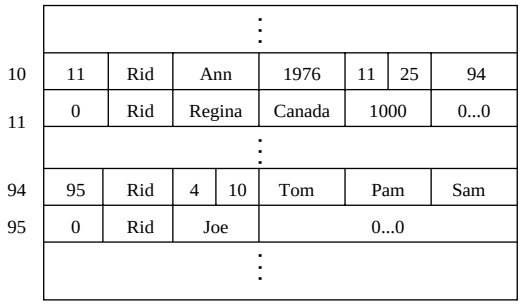


Figure 9. The updated fact.

3.1.3. Index file

The index file contains indexes which associate key values with fact node addresses in the data file. Its structure is shown in Figure 10. The relation header tables at the beginning of an index file holds all logical index record tables for every primary and secondary key in the persistent relations. This structure is adopted from PARODY without any change.

Relation header tables. Each relation is represented by a table of index header records. The first node in the file is the header table for the relation that has relation identification zero. That node is the first of a thread of index header nodes. The second node in the thread is the table for relation identification one and so on.

Index header records. Each relation header record is a table of index header records. There is an index header record for each key in the relation. The first one is for the primary key. The second one is for the first secondary key, and so on.

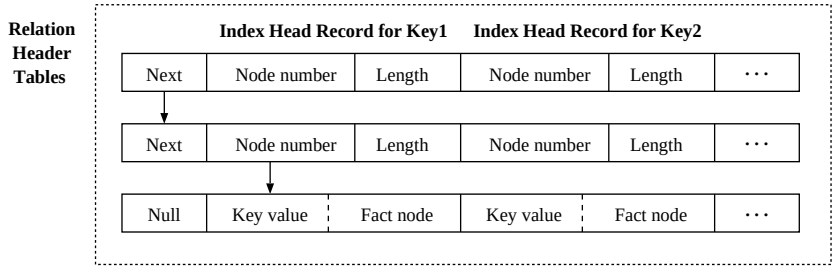


Figure 10. Index file.

Each index header record contains two data items: the node number of the root node for the index and the length in bytes of the index key value. The system assigns the root node number when the index is first built, and it extrapolates the key length from the first time the key is created.

3.1.4. System catalog files

PARODY does not support meta information. Thus it is furnished with the capability for meta information management in the system catalog files. The structure of catalog files is the same as the basic structure in Figure 6. Each system catalog file, however, has a different node size depending on its basic block. A basic block can be the description of a database, a relation, an attribute, a rule, or a rule literal. It takes up exactly one node in its corresponding file. Most kinds of blocks are connected with each other through node numbers.

Representation of schemas. In the Relationlog system, all relation schemas in a database are created as novel tree-like structures. Each attribute is identified by its unique node number. Attributes are connected to each other by pointers (node numbers) in the catalog file *attribute.sys*. For example, the Persons relation in Figure 2 is represented internally as in Figure 11.

The schema shows that each fact of relation Persons is a tuple. There are several attributes in this tuple, which are Name as a **string** type, Birthday as a tuple of three **integer** types: Year, Month, and Day, Parents as a set of **string**, LivesIn as a tuple of two **string** types: City and Country, and Earning as a list of **integer**.

Note that if the type of an attribute is tuple, set, list, or bag, then it has some sub-types as its children. Figure 12 shows a more complicated schema definition of the ROBOT relation in Figure 3.

Representation of rules. The Relationlog rules have been introduced in Section 2. Each rule has a few literals concerning relations or arithmetic and set-oriented operations. Each literal is called a query (rule) node in internal representation. Each query (rule) node has a few query (rule) units corresponding to the attributes if the unit is associated with one of the relations in the database. Note that a rule can be used as a query, query nodes, and units are used to stand for rule nodes, and units for the compatibility with queries. The units are connected to their relatives by node numbers in the tree-like structure like

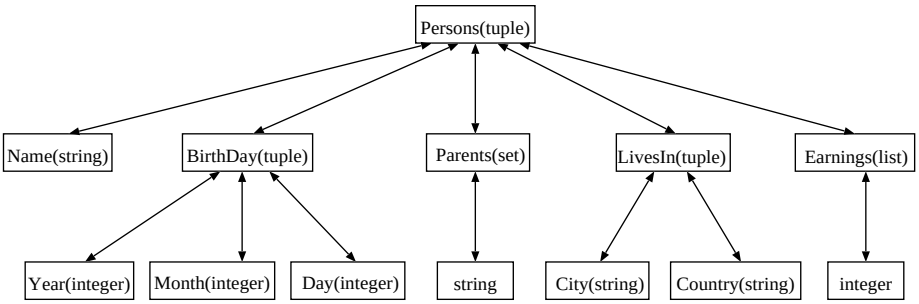


Figure 11. Schema structure for Persons.

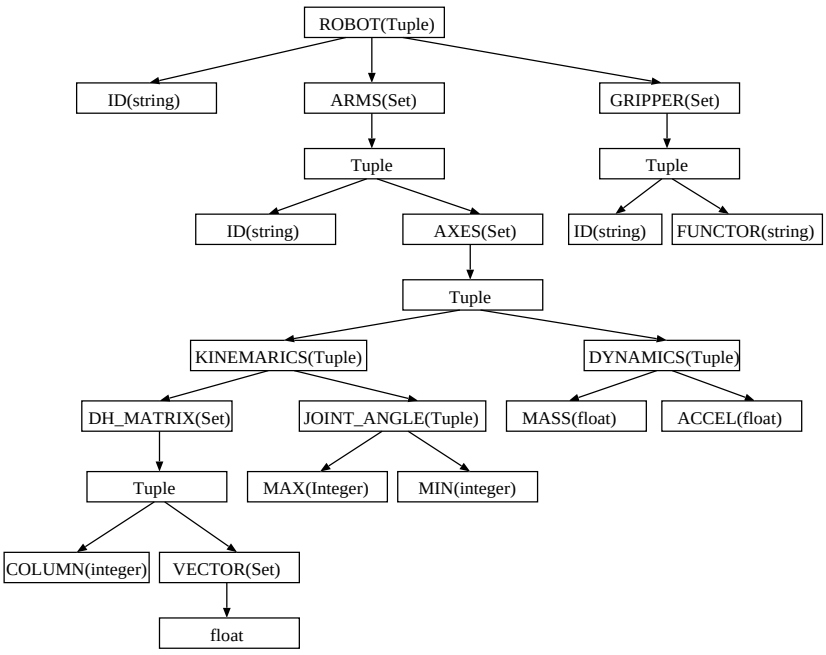


Figure 12. Schema structure for ROBOT.

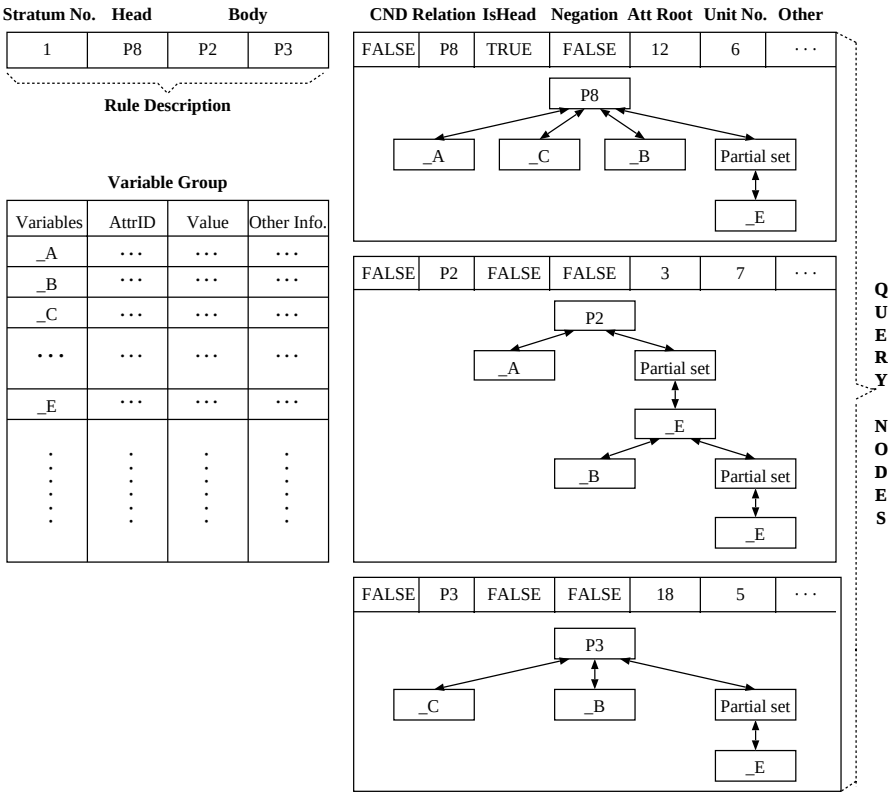


Figure 13. Internal rule representation.

schemas. Each unit can contain a constant or a variable pointer to VariableGroup, which contains all the variables in the rules of a database. However, if the query node is only a condition operation, a label is set to indicate a condition node.

Consider the following rule:

$$P_8(\_A, \_C, \_B, \langle \_E\rangle) :- P_2(\_A, \langle[\_B, \langle \_E\rangle]\rangle), P_3(\_C, \_B, \langle \_E\rangle)$$

Its internal representation is shown in Figure 13. This rule has three literals corresponding to three query nodes. The description of the rule includes the stratification layer number (Stratum No.), head, and body node numbers of the rule (for easier understanding, relation name is used instead). Each query node has a few members: CND indicating whether the node is a condition (TRUE for YES), relation node number Relation (relation name is still used), IsHead denoting the position of the node in a rule, Negation (TRUE) defining the negative literals, AttrRoot standing for the attribute root node

number for this query node, UnitNo. illustrating the unit number of the query units belonging to the query node, and some other bytes left for later use. A query unit is related to a variable or a constant. The variable in a query unit is represented as a pointer pointing to VariableGroup. To distinguish variables with the same name, some tags are used such as related attribute node number (AttrID), and other information (Other Info.), like rule number and so on. The Value column associates the real value for the variable in the query processing, which could also be a pointer. If the query node is a condition, the operator is indicated, and so are the operands (variables or constants).

In the example, it is assumed that the stratification layer of the rule is 1, and the attribute root node numbers for the three query nodes are respectively 12, 3, and 18. Then the description of the rule is P_8 for the head, and P_2 and P_3 for the body. In the real system, these IDs for the head and body are the node numbers for each query node. But for better illustration, the names are used in Figure 13 instead of the real node number. Three query nodes are needed with the relation name of P_8, P_2, and P_3. They are all non-condition nodes among which TRUE in IsHead field of P_8 indicates it is the head of the rule, while FALSE for others indicates the body of the rule. For the node of P_8, there are six query units corresponding to the head literal. There are 7 and 5 query units respectively for the two body nodes. In the rule, there are variables _A, _B, _C, and _E storing in VariableGroup, where the variable names are connected with their nodes, attributes and values. Because _E here represents a tuple, its Value field in VariableGroup contains the node number holding the whole tuple in case of evaluation.

Representation of catalogs. Each system catalog file has a different node size depending on its basic block. A basic block can be an attribute node, a query/rule node, and a query/rule unit node for schemas and rules discussed in the previous two sections. It can also be the description of a database and a relation. It takes up exactly one node in its corresponding file. The next node field of each node points to the next node or rule depending on the context. Each node may contain some pointers to other files. In *database.sys*, each node of the description of a database contains some pointers, actually node numbers, pointing to the description of its related relations and rules. It also stores VariableGroup for all the variables in the rules. In *relation.sys* and *rule.sys*, nodes of the relations and rules are connected to the next by node numbers. The description node of a relation contains information about the schema pointing to *attribute.sys* and other information like layer number and so on. Likewise, the description nodes of rules are hooked up to each other also by node numbers and contain pointers to their rule literals in *ruleunit.sys*.

Consider the Family database in Figure 4. The meta information concerning its relation schemas and rules is partly given in Figure 14. The node in *database.sys* contains the description of the database family denoting all the meta information of its relation schemas and rules. In *relation.sys* or *rule.sys*, each node stream contains the description of all relations or rules, such as Parentsof and Ancestorsof in *relation.sys*. There is a node number in a description node of a relation pointing to the root node of its schema in *attribute.sys*, through which all attributes can be accessed. Similarly a description node of a rule has the node numbers of its head and first body rule units, and thus points to its head and body array in *ruleunit.sys*.

In summary, when a database is activated (opened) in the Relationlog system, all attributes of domain types, schemas, rule nodes, and rules units included in this database are created as tree-like structures in main memory from the system catalog files in order to achieve high performance. Each block (node) is identified by its unique node number. Units are connected to each other by node numbers. The indexes are loaded in memory only in need for facts, whose representation is described in the following section.

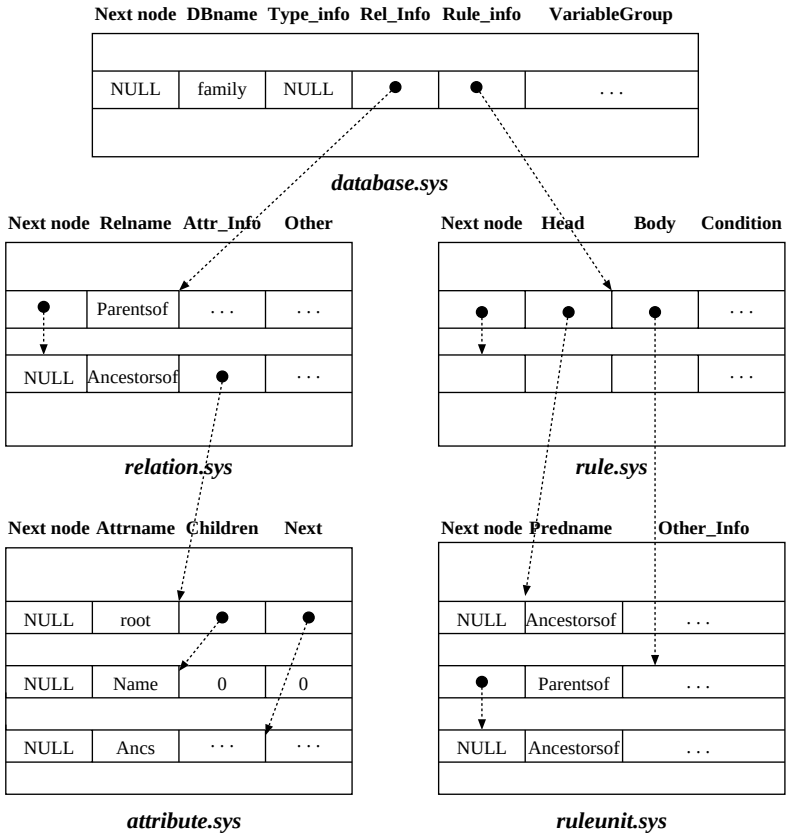


Figure 14. System catalog files.

3.2. Storage and update subsystem

The Storage and Update Subsystem is the lowest layer. It accepts the requests from the Data Management and Query System and responds to them with *TRUE* indicating jobs done, or *FALSE* denoting a failure along with some error message. According to the requests, it provides rapid access to domains, schemas, facts, indexes, and rules in the persistent database.

Generally speaking, the subsystem performs the following tasks:

1. memory buffer management, such as loading and moving data in and out of memory
2. updates of domains, schemas, facts, indexes, and rules
3. retrieval of attributes, facts, indexes, rules by their unique node numbers, which is much like most object-oriented DBMSs using oid to identify an object
4. supports for the B-tree and Hash indexes

5. supports for many different kinds of data types, such as atomic data types, *tuple* types, *set* types, *bag* types, and *list* types so that the user can declare and manipulate complex nested relations with these types.

3.2.1. Buffer Manager

Node numbers are used as pointers in the Relationlog system. They have to be translated into exact memory addresses for the accesses in memory. The Buffer Manager is in charge of address mapping that translates node numbers of attributes, facts, indexes, and rules into memory addresses and updates while cooperating with the Update Manager.

To make efficient use of each node, Relationlog adopts a technique similar to that used in Cache-Memory Mapping, called *Set Associative Mapping*. Each relation has its own buffer space for its own facts. The buffer space is composed of a number of memory pages, holding some memory slots. This manager is responsible for managing the space occupied by objects (including facts, attributes, indexes, and rules) requested by the applications. This means if an object is currently in main memory, then its memory address is returned to the application requesting it. Otherwise the Buffer Manager employs an LRU algorithm to drop some old objects that have not been used for a long time, and then loads this object into memory. In the case that the node to be dropped has been changed or deleted (marked by the Update Manager), the Buffer Manager physically changes or deletes the node stored on disk. The format of an object in its page buffer is similar to its disk format.

For example, suppose there are 8 slots in memory and 24 nodes on disk. The 8 slots are grouped into 2 sets: *Group0* and *Group1*. Similarly, the 24 nodes are divided into 6 sets: *G0*, ..., *G5*. In each set, there are 4 slots. According to the mapping scheme, *Group0* corresponds to *G0*, *G2*, *G4*, and *Group1* to *G1*, *G3*, *G5*. The mapping between memory slots and disk nodes is shown in Figure 15. If an application needs node 10, then $10 \ mod \ 4 = 2$ indicates that the node is in *G2*, while *G2* corresponds to *Group0*. Thus node 10 should be in memory *Group0*. At the same time, the Buffer Manager selects a slot in the set and sets *Node No.* to the corresponding node number and makes a mark in the *Tags* field. Assume slot 0 is the first free slot or the slot can be made available by using LRU, then slot 0 is the one chosen for loading.

3.2.2. Index Manager

The Index Manager maintains the B-tree indexes for the Relationlog relations (extensional and intensional) and Hash indexes for temporal relations. They are chosen because they are efficient and standard, and are used by most database systems. B-tree indexes reside both in memory and on disk and are used for defining and maintaining keys of each relation. They are stored in the index file of each database. Hash indexes are applied only in memory for temporal relations, which could be the differential relations of semi-naive evaluation or temporarily created views.

3.2.3. Update Manager

The Update Manager deals with the updates of these nodes that are for domains, schemas, facts, indexes, and rules. As an update may imply a query, the DML and DDL Managers consult with the

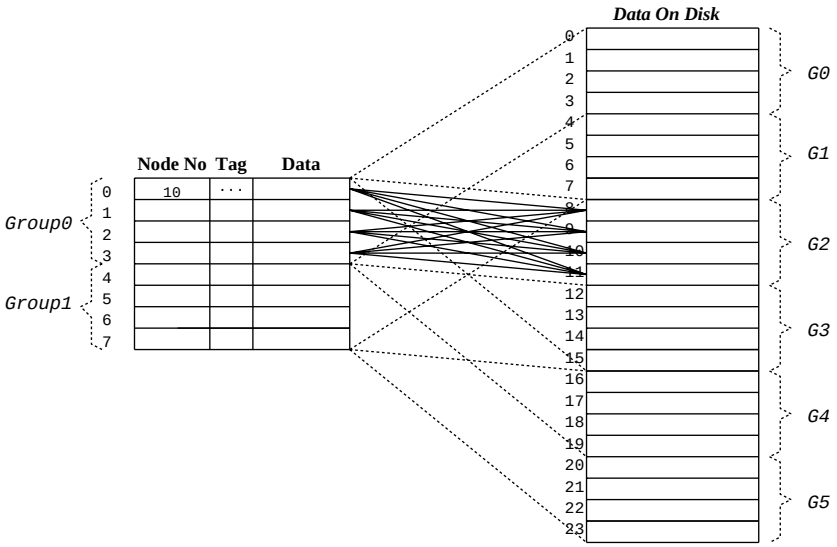


Figure 15. Mapping between disk and memory.

Query Manager to find the exact nodes to be updated and then send these node numbers to the Update Manager to perform the updates. The Update Manager then consults with the Buffer Manager and the Index Manager to get the nodes that need to be updated into memory (if it is not there). It also marks the nodes with a *changed* or *deleted* tag on its corresponding memory page slot so that the Buffer Manager can change or delete them when the session is over (the database is closed) or the nodes are dropped out from memory.

3.3. Data management and query subsystem

This section first describes the DDL Manager, DML Manager, and Query Manager. It also discusses the extended semi-naive algorithm and extended magic-set rule rewriting technique used in Relationlog.

3.3.1. DDL Manager

The DDL Manager processes all Relationlog DDL commands, maintains system catalogs about domains, schemas, relations, indexes, and rules, and answers all the type checking requests and rule applications from the DML Manager or the Query Manager. It also checks if all rules are stratified when a new view is created. As views can be materialized, the DDL Manager is also responsible for dropping those materialized intensional relations and the corresponding rules when a view is dropped. Figure 16 shows its four subparts: DDL Processor, Schema Manager, Type Checker, and Rule Manager.

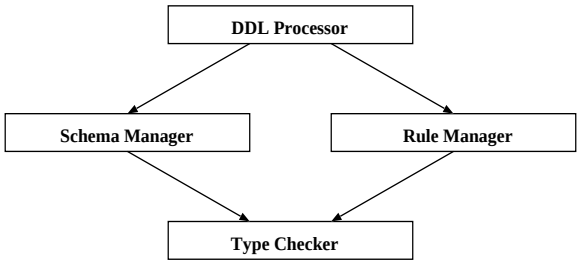


Figure 16. Components of the DDL Manager.

DDL Processor The Data Definition Language processor is responsible for pre-processing the user inputs, checking the validity and then translating them into internal expressions. It communicates with the Schema and Rule Managers to gain exclusive access of the schemas and rules, and makes the changes.

Schema Manager The Schema Manager is responsible for implementing all changes to schema definitions. It processes the internal schema update expressions from the DDL Processor. By further checking the syntax and semantics safety and soundness, the request with internal schema update expressions is proceeded or rejected. The Schema Manager keeps a version of the schema in the catalog files and tracks the changes of every attribute.

Type Checker The Type Checker is responsible for ensuring that the submitted queries and update expressions are legal. This involves attribute identifier resolution and type checking the attribute values. Example errors include unknown relations or attribute names, passing incompatible types to arithmetic operators, and malformed expressions. In Relationlog, types are checked each time a query is submitted. This ensures that, after modification of the schema, the Type Checker does not accept incorrect queries that have been generated by the older application programs. Timings of Relationlog operations indicate that type checking is only a small part of the overall cost of query evaluation.

Rule Manager The Rule Manager maintains the essential information concerning rules. It deals with the internal rule update expressions from the DDL Processor. First of all, it consults with the Type Checker to make sure that the expressions are well-typed. Then the safety and acceptability regarding the requests are checked. After that, the internal expressions are optimized. If everything goes well with the checking and optimization, the request is committed. Otherwise it fails. The Rule Manager also maintains a hard copy of the rules in the system catalog files and makes safe changes on it.

3.3.2. DML Manager

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 the materialized views that are dependent on the updated relation, if any.

3.3.3. Query Manager

The Query Manager is responsible for data retrieval and rule evaluation pertaining to the facts stored extensionally in the database or defined intensionally by rules. It first translates a query into its internal expressions, optimizes them based on the conditions the query contains and then uses different evaluation strategies such as matching, semi-naive bottom-up evaluation with rule ordering, and magic-set rule rewriting techniques to find the results.

Internal representation of queries Because each variable is associated with an attribute and so is each known value, Relationlog adopts an internal tree-like structure, and each node in the tree is called a query unit, for each query literal related to a relation. Thus the Type Checker can easily manipulate the type checking and the communication with the Schema Manager. Each query unit is connected with its neighbors and children through pointers, which simulates the representation of the relation schemas. The internal representation of query expressions is the same as that of the units in a rule illustrated in Section 3.1.4. The difference between them is that all the meta information (nodes and units) of rules is stored in the catalog files, while that of queries is temporally in memory.

Matching Unlike other deductive database systems, Relationlog allows some intensional facts to be persistent on disk. Also, it keeps non-persistent intensional facts inferred using rules in memory if the memory space permits to speed up query answering and avoid redundant computation. Therefore, the result to the user query may already be on disk or in main memory without any need for additional evaluation. In this case, Relationlog simply uses matching to find the result to the user query.

Extended semi-naive technique Relationlog uses an extended semi-naive algorithm coupled with extended magic-set rule rewriting. In this section, we describe the extended semi-naive algorithm.

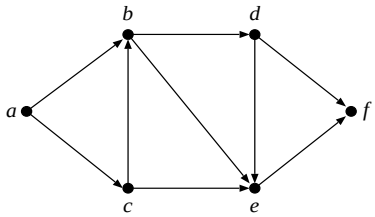
The semi-naive algorithm for Relationlog uses two operators. One is the grouping operator, denoted by $\uplus$, which is used to group sets of compatible tuples. The following is an example:

$$\uplus\,\{(a, \langle b\rangle), (a, \langle c\rangle)\} = \{(a, \langle b, c\rangle)\}$$

The other is the difference operator, denoted by $-$, which is used to get the difference of two nested relations. The following is an example:

$$\{([a, \langle b, c\rangle, \langle e, f\rangle])\} - \{([a, \langle c\rangle, \langle e\rangle])\} = \{([a, \langle b\rangle, \langle f\rangle])\}$$

The following graph represents the edges between nodes:



In Relationlog, we can use a nested relation *edge* which is also known as an adjacency list to store this graph as follows:

edge	*from*	*to*
	a	$\{b, c\}$
	b	$\{d, e\}$
	c	$\{b, e\}$
	d	$\{e, f\}$
	e	$\{f\}$

The transitive closure can be expressed using the following Relationlog rules:

$$(R1): \quad path(\_X, \langle \_Y \rangle) :\!- edge(\_X, \langle \_Y \rangle)$$
$$(R2): \quad path(\_X, \langle \_Y \rangle) :\!- path(\_X, \langle \_Z \rangle), edge(\_Z, \langle \_Y \rangle)$$

where $\langle \_Y \rangle$ and $\langle \_Z \rangle$ are partial set terms which are used in two different ways: to denote part of a set when in the body of the rule and to derive part of a set to be grouped together when in the head of the rule.

The semi-naive algorithm for the intensional relation *path* is as follows:

$$\begin{cases} \triangle path^1(\_X, \langle \_Y \rangle) & \leftarrow edge(\_X, \langle \_Y \rangle) \\ path^1 & := \uplus \triangle\, path^1(\_X, \langle \_Y \rangle) \end{cases}$$

$$\begin{cases} path^{i+1}_{temp}(\_X, \langle \_Y \rangle) & \leftarrow edge(\_X, \langle \_Z \rangle), \triangle path^i(\_Z, \langle \_Y \rangle) \\ \triangle path^{i+1}(\_X, \langle \_Y \rangle) & := path^{i+1}_{temp} - path^i \\ path^{i+1} & := \uplus\, (path^i \cup \triangle path^{i+1}(\_X, \langle \_Y \rangle)) \end{cases}$$

where $path^{i+1}_{temp}$ contains the facts inferred at the $i+1$ stage, $\triangle path^{i+1}$ contains only the facts in $path^{i+1}_{temp}$ that are not in $path^i$ (i.e. new facts), and $path^{i+1}$ contains all the facts in *path* that are inferred at or before the $i+1$ stage. The algorithm stops when $\triangle path^{i+1}$ is empty.

Using the above algorithm, we obtain the following nested relation in Relationlog:

path	*from*	*to*
	b	$\{d, e, f\}$
	c	$\{b, d, e, f\}$
	d	$\{f\}$
	e	$\{f\}$

Extended magic-set strategy The magic-set technique simulates in semi-naive bottom-up evaluation the pushing of selections that occurs in top-down approaches. Its performance can rival the efficiency of the top-down techniques. Here, we assume that readers are familiar with adorned rules and magic-set rewriting and thus only give a brief introduction to the notations used in Relationlog.

Table III. Extened adornments in Relationlog.

Argument	Adornment	Note
a	b	a is a constant
X	f	X is an atomic variable
$\langle a, b, c\rangle$	P^{bbb}	a, b, c are constants
$\langle a, b, X\rangle$	P^{bbf}	a, b are constants and X is an atomic variable
$\{a, b, c\}$	C^{bbb}	a, b, c are constants
$\{X, Y, Z\}$	C^{fff}	X, Y, Z are atomic variables
(a, X, Y)	$bC^{f}T^{f}$	X is a set variable and Y is a tuple variable
$(a, \{X, Y\})$	bC^{ff}	X, Y are atomic variables
$(a, \langle b, c\rangle, (d, e))$	$bP^{bb}T^{bb}$	a, b, c, d, e are constants

In Relationlog, there is a fixed order of the arguments for each predicate. The arguments can be of four kinds: atomic term, complete set term, partial set term, and tuple term. As Relationlog is a typed language, the kinds of arguments allowed for each predicate are in fact known, based on the schema.

There are different levels in relations in Relationlog. Each level represent sets or tuples nested in up-level sets or tuples. For the ROBOTS relation in last section, there are four levels. The first is ROBOTS(Id, Arms,Grippers); the second is Arms:{[Id,Axes]} and Grippers:{[Id,Function]}; the third is Axes:{[Kinematics, Dynamics]}; and the fourth is Db_matrix:{[Column,Vector]} in Axes of Arms.

In order to represent the bound or free information to a deeply nested level, an extended adornment for n-ary predicate p in Relationlog is a string of length n on the alphabet $\{b, f, C, P, T\}$, where b is for atomic argument and stands for bound; f is for atomic argument and stands for free; C is for complete set term; P is for partial set term; and T is for tuple term. Moreover, for a complete set term, partial set term or tuple term, we use another string on the alphabet $\{b, f, C, P, T\}$ as a superscript to represent the status of arguments in it.

Table III shows examples of extended adornments in Relationlog.

Based on these extended adornments, we extend the magic set approach [16,18] to deal with nested sets and tuples in Relationlog so that if the query involves some constant, only relevant intensional facts will be generated. See Reference [44] for details.

4. COMPARISON WITH OTHER SYSTEMS

In this section, we discuss the major differences and similarities between Relationlog and other related deductive database systems.

In the past decade, a large number of various deductive database languages have been proposed. Many of them have been implemented, which include Glue-Nail [45], LOLA [46], XSB [27], Aditi [28], LogicBase [29], Declare/SDS [30], LDL [38], CORAL [39], COL [34], CORAL++ [47],

ROCK & ROLL [48], Florid [49], ROL [50] and ROL2 [51]. These systems can be classified into three kinds:

1. Datalog based such as Glue-Nail, XSB, Aditi, LORA, and LogicBase
2. complex object based such as LDL, COL, and CORAL
3. object-oriented such as CORAL++, ROCK & ROLL, Florid, ROL, and ROL2.

Relationlog belongs to complex object based. It does not support many key object-oriented features. From the language point of view, Relationlog combines the best of LDL, COL, and Hilog [35]. Its partial set terms can be used to directly access deeply nested tuples and sets in relations as if they are normalized and to inference and construct complex objects. Thus, it is more powerful and flexible than other complex object based deductive languages. See Reference [37] for details.

From the implementation point of view, most of these implemented deductive database systems are only main-memory database systems that require everything to be in main memory. Some of them provide interface to external persistent stores. Relationlog is a real persistent database system that supports persistent schema, rules, and facts directly. It does not require everything to be in main memory. Its storage and update subsystem takes care of the memory space. If the memory space is used up, it removes the least recently used data out of memory before it loads new data.

Unlike object-oriented deductive systems that rely on object identifiers to support complex objects such as ROL and ROL2, Relationlog directly supports the storage of nested relations/complex objects. The persistent file structure of Relationlog is quite unique compared to other approaches in the implementation of nested relations such as Verso [52], DASDBS [53], and ANDA [54,55].

LDL, CORAL, Glue-Nail, Aditi, and LOLA use the bottom-up evaluation approach with magic-set rewriting at compile time. They only allow static rules that cannot be queried or updated, and require the user to provide query forms so that the rules can be compiled based on the query forms in advance. For a predicate with n arguments, there are 2^n possible query forms to be specified. Some query forms are not allowed because of the limitation of evaluation strategies. If a query cannot match any query form, it will be rejected. This is burdensome from the user's point of view. They also discard intensional objects so that if the same query is asked again, it will be evaluated from the beginning again.

Relationlog supports dynamic rules and allows rules to be queried and updated. Most importantly, it does not require query forms. The user can issue any syntactically correct queries, and Relationlog dynamically rewrites rules relevant to the queries using an extended magic-set strategy. Relationlog also keeps intensional facts in main memory to speed up query answering and avoid redundant computation. In fact, Relationlog incorporates many features of the ROL system [50].

5. CONCLUSION

In this paper, we have described the Relationlog system prototype. A complete implementation as described in this paper has been developed. The system is available from the web page: `http://www.cs.uregina.ca/~mliu/RLOG`. Several applications based on Relationlog have been developed as graduate student research projects. A major one is using Relationlog to support CAD [56].

The main novel feature of Relationlog is that it is the first system that supports the storage and inference of data with complex structures.

The most obvious limitation of the current Relationlog implementation is the lack of concurrent access to the database. While there is little question on the technical feasibility and desirability of such a system, building it is not our top priority as we are more concerned with the storage and inference of data with complex structures. Another limitation is that Relationlog does not support null values. There is no doubt that null needs to be addressed in any real database system. In fact, supporting null in a non-deductive database system is straightforward. However, it is still not clear how to perform deduction in a deductive database when null is allowed. We would like to investigate how to support null in Relationlog.

As the current implementation of the Relationlog system is built on top of PARODY [40], Relationlog performs reasonably well for small and medium databases and relatively slowly for large ones due to the limitation of PARODY itself. We would like to overcome this limitation in our further research.

We are currently extending Relationlog into an object-relational deductive system by adding object identifiers, inheritance, and rule-based functions and procedures. We are also developing OQL [57], extended relational algebra, and calculus [8,58,59] interfaces on top of it in order to make it a useful tool for teaching database courses. We are also exploring other query processing strategies for the Relationlog system and extending it into a fully-fledged system.

ACKNOWLEDGEMENTS

The author acknowledges the contributions of the following people to the Relationlog system: Riqiang Shan, Tao Guan, Shilpesh Katragadda, Ming Zhao, Mingyuan Deng, and Sudha Srinivas. Thanks also to the referees for their valuable comments and suggestions. This work has been partly supported by the Natural Sciences and Engineering Research Council of Canada (NSERC grant OGP0193553-1996).

REFERENCES

1. Abiteboul S, Fischer PC, Fischer PC (eds.). *Proceedings of the International Workshop on Theory and Applications of Nested Relations and Complex Objects in Databases*, Darmstadt, Germany (*Lecture Notes in Computer Science* 361). Springer-Verlag, 1987.
2. Abiteboul S, Beeri C. The power of languages for the manipulation of complex values. *The VLDB Journal* 1995; **4**(4):727–794.
3. Colby LS. A recursive algebra for nested relations. *Information Systems* 1990; **15**(5):567–582.
4. Hull R. A survey of theoretic research on typed complex database objects. *Databases*, Pardaens F (ed). Academic Press, 1987; 193–256.
5. Levene M, Loizou G. Semantics for null extended nested relations. *ACM Transactions on Database Systems* 1993; **18**(3):414–459.
6. Meral Ozsoyoglu Z, Yuan L-Y. A new normal form for nested relations. *ACM Transactions on Database Systems* 1987; **12**(1):111–136.
7. Roth MA, Korth HF, Batory DS. SQL/NF: A query language for ¬1NF relational databases. *Information Systems* 1987; **12**(1):99–114.
8. Roth MA, Korth HF, Silberschatz A. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems* 1988; **13**(4):389–417.
9. Carey M, DeWitt D, Vanderberg S. A data model and query language for EXODUS. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Chicago, IL, 1988; 413–423.
10. Fishman DH, Beech B, Cate HP, Chow EC, Connors T, Davis JW, Derrett N, Hoch CG, Kent W, Lyngbaek P, Mahbod B, Neimat MA, Ryan TA, Shan MC. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems* 1987; **5**(1):48–69.

11. Ishikawa H, Suzuki F, Kozakura F, Makinouchi A, Miyagishima M, Izumida Y, Aoshima M, Yamane Y. The model, language, and implementation of an object-oriented multimedia knowledge base management system. *ACM Transactions on Database Systems* 1993; **18**(1):1–50.
12. Kim W. *Introduction to Object-Oriented Databases*. The MIT Press, 1990.
13. Lecluse C, Richard P. The O_2 database programming language. *Proceedings of the International Conference on Very Large Data Bases*, Amsterdam, The Netherlands. Morgan Kaufmann Publishers, Inc., 1989; 411–422.
14. Ceri S, Gottlob G, Tanca T. *Logic Programming and Databases*. Springer-Verlag, 1990.
15. Ullman JD. *Principles of Database and Knowledge-Base Systems*, vols. 1 & 2. Computer Science Press, 1989.
16. Bancilhon F, Maier FD, Sagiv Y, Ullman JD. Magic sets and other strange ways to implement logic programs. *Proceedings of the ACM Symposium on Principles of Database Systems*, Cambridge, MA, 1986; 1–16.
17. Bancilhon F, Ramakrishnan R. An amateur's introduction to recursive query processing strategies. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., 1986; 16–52.
18. Beeri C, Ramakrishnan R. On the power of magic. *Journal of Logic Programming* 1991; **10**(3,4):255–299.
19. Ioannidis YE, Ramakrishnan R. Efficient transitive closure algorithms. *Proceedings of the International Conference on Very Large Data Bases*, Los Angeles, CA, U.S.A. Morgan Kaufmann Publishers, Inc., 1988; 382–394.
20. Jiang B. A suitable algorithm for computing partial transitive closures. *Proceedings of the International Conference on Data Engineering*, Kobe, Japan. IEEE Computer Society, 1990; 264–271.
21. Mumick IS, Finkelstein SJ, Pirahesh H, Ramakrishmnan R. Magic conditions. *ACM Transactions on Database Systems* 1996; **21**(1):107–155.
22. Sacca D, Zaniolo C. Magic counting methods. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987; 49–59.
23. Ullman JD. Bottom-up beats top-down for datalog. *Proceedings of the ACM Symposium on Principles of Database Systems*, Philadelphia, PA, 1989; 140–149.
24. Morris K, Ullman JD, Gelder AV. Design overview of the Nail! system. *Proceedings of the International Conference on Logic Programming*, London, England. MIT Press, 1986; 554–568.
25. Freitag B, Schutz H, Specht G. LOLA – A logic language for deductive databases and its implementation. *Proceedings of the International Symposium on Database Systems for Advanced Applications (DASFAA '91)*, Tokyo, Japan, 1991; 216–225.
26. Derr M, Morishita S, Phipps G. Design and implementation of the Glue-Nail database system. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., 1993; 147–167.
27. Sagonas K, Swift T, Warren DS. XSB as an efficient deductive database engine. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Minneapolis, MN, 1994; 442–453.
28. Vaghani J, Ramanohanarao K, Kemp DB, Somogyi Z, Stuckey PJ, Leask TS, Harland J. The aditi deductive database system. *The VLDB Journal* 1994; **3**(2):245–288.
29. Han J, Liu L, Xie Z. LogicBase: a deductive database system prototype. *Proceedings of the International Conference on Information and Knowledge Management*, Gaithersburg, MD. ACM, 1994; 226–233.
30. Kiebling W, Schmidt H, Straub W, Dunzinger G. DECLARE and SDS: Early efforts to commercialize deductive database technology. *VLDB Journal* 1994; **3**(2):211–243.
31. Ramakrishman R, Ullman JD. A survey of deductive database systems. *Journal of Logic Programming* 1995; **23**(2):125–150.
32. Naqvi S, Tsur S. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, 1989.
33. Kuper GM. Logic programming with sets. *Journal of Computer and System Sciences* 1990; **41**(1):44–64.
34. Abiteboul S, Grumbach S. COL: A logic-based language for complex objects. *ACM Transactions on Database Systems* 1991; **16**(1):1–30.
35. Chen Q, Chu W. HILOG: A high-order logic programming language for non-1NF deductive databases. *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, Kim W, Nicolas JM, Nishio S, (eds.). North-Holland, 1989; 431–452.
36. Liu M. Relationlog: A typed extension to datalog with sets and tuples. *Journal of Logic Programming* 1998; **36**(3):271–299.
37. Liu M. Deductive database languages: Problems and solutions. *ACM Computing Surveys* 1999; **30**(1):27–62.
38. Chimenti D, Gamboa R, Krishnamurthy R, Naqvi S, Tsur S, Zaniolo C. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering* 1990; **2**(1):76–90.
39. Ramakrishnan R, Srivastava D, Sudarshan S, Seshadri P. The CORAL deductive system. *VLDB Journal* 1994; **3**(2):161–210.
40. Stevens A. *C++ Database Development* (2nd edn.). MIS Press, 1994.
41. Liu M. Extending datalog with declarative updates. *Proceedings of the 11th International Conference on Database and Expert System Applications (DEXA 2000)*, Greenwich, UK, 4–8 September 2000 (*Lecture Notes in Computer Science*). Springer-Verlag, 2000; 752–763.

42. Shan R, Liu M. Introduction to the Relationlog system. *Proceedings of the 6th International Workshop on Deductive Databases and Logic Programming (DDLP '98)*, Manchester, UK, 20 June 1998; 71–83.
43. Liu M, Shan R. The design and implementation of the Relationlog deductive database system. *Proceedings of the 9th International Workshop on Database and Expert System Applications (DEXA Workshop '98)*, Vienna, Austria, 24-28 August 1998. IEEE-CS Press. 1998; 856–863.
44. Liu M. Query processing in Relationlog. *Proceedings of the 10th International Conference on Database and Expert System Applications (DEXA '99)*, Florence, Italy, 30 August–3 September 1999 (*Lecture Notes in Computer Science*, vol. 1677). Springer-Verlag, 1999; 342–351.
45. Derr MA, Morishita S, Phipps G. The Glue-Nail deductive database system: Design, implementation, and evaluation. *VLDB Journal* 1994; **3**(2):123–160.
46. Zukowski U, Freitag B. The deductive database system LOLA. *Proceedings of the International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR '97)*, Dagstuhl, Germany (*Lecture Notes in Computer Science*, vol. 1265). Springer-Verlag, 1997; 376–387.
47. Srivastava D, Ramakrishnan R, Srivastava D, Sudarshan S. CORAL++: Adding object-orientation to a logic database language. *Proceedings of the International Conference on Very Large Data Bases*, Dublin, Ireland. Morgan Kaufmann Publishers, Inc., 1993; 158–170.
48. Barja ML, Fernandes AAA, Paton NW, Williams MH, Dinn A, Abdelmoty AI. Design and implementation of ROCK & ROLL: a deductive object-oriented database system. *Information Systems* 1995; **20**(3):185–211.
49. Frohn J, Himmeröder R, Kandzia P, Lausen G, Schlepphorst C. Florid: A prototype for F-logic. *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society, 1997; 583.
50. Liu M. The design and implementation of the ROL deductive object-oriented database system. *Journal of Intelligent Information System* 2000; **15**(2):121–146.
51. Liu M. ROL2: A real deductive object-oriented database language. Submitted for Journal Publication, 2000.
52. Scholl M, Abiteboul S, Bancilhon F, Bidoit N, Gamerman S, Plateau D, Richard P, Verroust A. VERSO: A database machine based on nested relations. *Proceedings of the International Workshop on Theory and Applications of Nested Relations and Complex Objects in Databases*, Darmstadt, Germany (*Lecture Notes in Computer Science*, vol. 361). Springer-Verlag, 1987; 27–49.
53. Schek HJ, Paul HB, Scholl MH, Weikum G. The DASDBS project: Objectives, experiences, and future prospects. *IEEE Transactions on Knowledge and Data Engineering* 1990; **2**(1):25–43.
54. Deshpande A, Gucht DV. A storage structure for nested relational databases. *Proceedings of the International Workshop on Theory and Applications of Nested Relations and Complex Objects in Databases*, Darmstadt, Germany (*Lecture Notes in Computer Science*, vol. 361). Springer-Verlag, 1987; 69–83.
55. Deshpande A, Gucht DV. An implementation for nested relational databases. *Proceedings of the International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers, Inc., 1988; 76–87.
56. Katragadda S. DrawCAD: Using advanced database technologies for CAD. *M.Sc Thesis*, University of Regina, 1999.
57. Cattell RGG (ed.). *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann, 1996.
58. Abiteboul S, Bidoit N. Non first normal form relations: An algebra allowing data restructuring. *J. Computer and System Sciences* 1986; **33**(3):361–393.
59. Colby L. A recursive algebra and query optimization for nested relations. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Portland, Oregon, 1989; 124–138.

