The Design and Implementation of an Extendible Deductive Database System

J. Kiernan, C de Maindreville, E. Simon

I.N.R.I.A Rocquencourt 78153
Le Chesnay, FRANCE

Abstract:

This paper presents the design and implementation of a deductive database system. RDL1, is the production rule language for this system. This language is an extension of logic based languages towards the support of updates and complex domains. Different features of this language are described with its applications. The general architecture of the system is presented highlighting integration with a relational database system. Then, three original features of the system are discussed : extensibility through the definition of end user data types, query optimization techniques, and the extensibility of the control strategy. Future improvements and extensions conclude the paper.

key words : deductive database system, expert database system, production rules, abstract data types, integration, extensibility.

1. INTRODUCTION AND MOTIVATIONS.

In this section we present the main motivations that led to the design of the RDL1 deductive database system. These motivations are supported by the fact that it is now widely accepted that future database applications, including geographical databases, manufacturing and communications, will require some kind of rule based reasoning.

The deductive database area is primarily concerned with the study of the logic programming as a way of querying a database. A deductive database consists of an extensional database (EDB) and a set of derived predicates, defined using rules, called intensional database (IDB). A logic query is represented by a set of rules defining derived predicate. The DATALOG query language (a pure Horn clause language) is a toy representative of logic-based query languages. A lot of efforts has been devoted to its optimization. Recently, many proposals have emerged for extensions of DATALOG with increased expressive power, providing forms of non monotonic logic reasoning. One such extension, called DATALOG^{neg} allows negative literals in the body of rules. Further extensions allow negative litterals in the body and heads of rules, negation in the heads of rules are interpreted as deletions. This allows invalidating a previously asserted fact, which is a key aspect of database updates. We shall refer to the last extension as DATALOG with negation and updates.

In [Simon88], we gave a formal description of such a language, called RDL1, which is essentially a production rule language as for instance OPS5 [Brownston85]. A rule in this language consists of a conditional part which is a relational calculus expression, and a consequent part which is a sequence of insertions and deletions of tuples in deduced relations. The main reasons for implementing such a language are: (i) as a query language, the update primitives in the rules make the language very expressive and flexible, (ii) the same language can also be used as an update language, (for instance to deal with triggers), if updates to base relations are allowed in the rules. Recently, a similar language has been independently proposed for expert database systems [Delcambre88].

The purpose of this paper is to present the design of the RDL1 system. The structure of the paper is the following. In section 2, we first present the data model we use. This model is an extension of the relational model towards user defined types. The syntax and the semantics of the production rule language is then presented. Section 3 presents the execution model used to run the rules on a DBMS. Section 4 gives the general architecture of the system. Two main features are then described : (i) the query optimization techniques and (ii) the conflict resolution strategies used by the rule evaluator. Section 5 describes the implementation of the rule evaluator. The conclusion gives a first analysis of the system and points out improvements to achieve better performances.

2. THE RULE BASED LANGUAGE

2.1. Data model

We consider an extension of the relational model that extends the notion of domain to User defined Data Types. This approach is in the same stream as other work [Osborn86, Stonebraker86, Wilms88]. Relations are built from the predefined data types (also called primitive domains), and from user defined data types (also called complex domains). In the next section, we introduce the notion of user defined data types. Then, we present user-defined functions and their properties.

2.1.1. User-defined data types

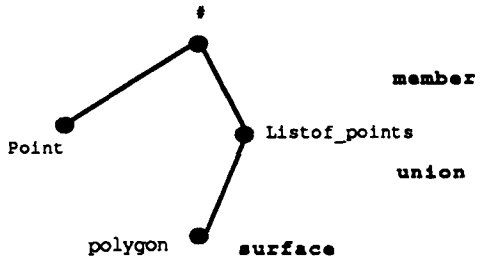
The data model we use extends the relational model with User Defined data Types (UDT). We assume a set of basic types from which more complex data types are constructed. The basic types are booleans, integers, real numbers and character strings. In the current version of our system, UDT are described in a Lisp language [Gardarin89]. UDT can be defined using the type constructors list and vector applied to basic types. Then, previously defined UDT can be used to build new data types. In addition, UDT are described in a "is-a" hierarchy and UDT operators can be inherited along the hierarchy. The extension of the base set of types together with the UDT form the set of domains from which tuples and relations can be built, [Gardarin89].

A specific primitive (function) in the Lisp language is used to define a new data type. The general syntax of this primitive called DD (for Define Domain) is:

```
(DD#:<function_name>(<parameter>)
          (<function_body>))
```

where function_name is the name of the UDT. It is defined as a path from the upper node of the "is_a" hierarchy to a current node. Each instance of the UDT will be bound to the parameter. The "function_body" is the code used to verify if the parameter value qualifies as an instance of the data type.

Defining a new UDT implies registering the corresponding DD function with the DBMS, that is to store the function definition into a database relation. The DBMS will later use the UDT definition as a domain integrity constraint to check that the new instances (for insertions or updates) of this type are consistent with the definition of the data type. The "is_a" hierarchy of UDT is then defined as follows. The semantics we give to type inheritance is set-inclusion. That is, the set of all instances of a subtype is included in the set of instances of its supertype. The common properties of several related types are abstracted out as another type. The related types are then defined as having the underlying type as their representation. Operations over the related types that are associated with the common representation type can then be identified and defined as part of the operations available on the new type. This saves rewriting these operations for each new subtype. For instance, the member and intersection operators for listof_pts can also be used for polygon. The diagram below represents a hierarchy of UDT.



2.1.2. User-defined functions

Operations over UDT can be defined by means of User Defined Functions (UDF). In the Lisp language, the standard DE and DF functions are used to define UDF. The general syntax of the function is:

```
(DE  #:(T1  ...Tn)  <function_name>  (par1
...parn)(<function_body>))
```

where T1 to Tn are the respective types of the parameters par_1 to par_n. UDF can be used in several ways. First, they enlarge the scope of functions already available in the DBMS. For instance, in the comparison operation: `surface (polygon) = 10.5` , the surface function will return a real value that will be passed to the DBMS to evaluate the predicate. The functions can also be used to directly manipulate UDT since standard DBMS operators (e.g., =, <) cannot be applied to them. In this case a function will return as a result an instance of a UDT. All the user-defined functions over a super type can be inherited by a subtype. For example, the UDT rectangle defined as a subtype of polygon inherits the above surface function. Thus, the surface function applied to a triangle will be allowed by the system.

Now, assume a surface function is defined for rectangles. The system will perform dynamically a selection of the right function according to the type of the arguments by traversing the hierarchy of types. If a column only stores values of the same type, then the right function can be selected once for all tuples.

Once a user has defined new data types and functions, their definition can be registered in the common UDT and UDF relation using the SAVE function. SAVE registers a particular function, its

result type, and its code with the database. The general syntax of the command is:

```
(SAVE #:argument1 #:argument2)
```

The first argument is the name of the function (or data type) and the second one is the type of the result returned by the function.

2.2. Syntax and semantics of rules

2.2.1 Syntax of rules.

An RDL1 program is composed of a set of if-then rules called productions that make up the rule base. The two key concepts of the language are the notions of condition and action.

The Right-Hand Side (RHS) of a production, (corresponding to the then part of the rule), is a set of actions. There are two elementary actions, denoted "+" and "-". The update action "+" takes a ground fact (i.e., a constant tuple) and maps a database state into another state which contains this fact. On the contrary, the action "-" takes a fact and deletes it from a relation.*Parameterized actions* can be specified by using as argument of the action a tuple containing either constants or variable terms. A *multiple action* consists of a sequence of elementary or parameterized actions.

The Left-Hand Side (LHS) of a production, (corresponding to the if part of the rule), is a formula of the tuple relational calculus. It consists of the conjunction of a *range definition part* defining tuple variables over relations and a *sub-formula* corresponding to a logical condition over the tuple variables. The figures below summarize the syntax of the rules.

```
<LHS> ::= <range condition> AND <sub-formula>
<range condition> ::= <range predicate>
                      NOT <range predicate>
                      <range condition> AND <range condition>

<sub-formula> ::= <expression>
                  NOT <expression>
                  <sub-formula> AND <sub-formula>

<expression> ::= <quantification>
                 (<predicate exp>)

<quantification> ::= EXISTS <var> IN <relation> <quantification>
                     EXISTS <var> IN <relation> (<predicate exp>)
                     FOREACH <var> .....
```

LHS of a rule

In the above table, we give a BNF syntax of the LHS of a rule. The non terminal symbol <predicate exp> denotes a formula of comparison predicates put in a conjunctive normal form. The figure below gives the syntax of the RHS of a rule. We only specify the syntax of elementary and parameterized actions; a multiple action consists of a sequence of these actions.

<action argument> ::= (<att>=<term> [,<att>=<term>...])
<term> ::= <constante> \ <var>.<att>

insertion	+ R (<action argument>)
deletion	-R (<action argument>)
update	-/+ R (<var>; <action argument>)

2.2.2 Semantics of rules

We described in the previous section the semantics assigned to an elementary action which is an insertion or a deletion of one tuple in a relation. A sequence of elementary actions is not interpreted as a sequential execution (i.e., a nested execution) of each action from left to right. Rather, a multiple action is seen as an atomic database update that maps a database state into a new one where the effect of all the actions have been taken into account. The effect of an elementary action + `Result` (a, b) is to assert that `Result` (a, b) must be present in the relation. Similarly, the action - `Result` (a, b) is to assert that the tuple (a, b) must not be present in the relation `Result` . Finally, the effect of the multiple action -`Result` (a, b) + `Result` (a, b) gives raise to a null action and the database state remains unchanged.

The evaluation of a LHS returns a set of elements matching the specified condition. This set is called the *matching set*. Each element in the matching set is an answer tuple to the query:

$$Q = \{t1, ..., tn \mid LHS\ (t1, ..., tn)\ \}$$

where t1, ..., tn are all the free variables that appear in the LHS of the rule. An element (a_1, ..., a_n) is in the matching set if and only if when each variable ti, $1 \le i \le n$, is replaced by a_i, the LHS of the rule becomes true. For instance, the evaluation of: R (x) and P (y) and x.att1 = y.att2 will return a matching set described by a relation whose schema is the union of the schema of R and the schema of P . This matching set is the answer to the query:

Q1 = {x.*, y.* | R (x) and P (y) and x.att = y.att }

When the matching set of a rule is not empty, the rule is said to be *relevant*. The semantics assigned to a rule is then as follows. First, each parametrized action + R (t_i) (or - R (t_i)) is interpreted as a sequence of elementary actions + R(c1) ... + R(ck) where each cj is an answer tuple to the query:

Q = project (ti; matching_set).

For instance, assume the action + result (x.att1, x.att2, y.att3) in the RHS of a rule whose LHS is : R (x) and P (y) and x.att1 = y.att1. Then, this action is interpreted as a sequence of elementary actions + result (a1) ... + result (an) where each aj is an answer tuple to the query:

Q2 = project (x.att1, x.att2, y.att3 ; Q1)

When each parameterized action in the RHS of a rule has been replaced by a sequence of elementary actions with respect to a matching set, we call it an *instantiated RHS*. Then, the effect of the rule is determined by the effect of its corresponding instantiated RHS with respect to the matching set of the rule. An instantiated RHS is said *effective* if it leads to a new database state. The only *firable* rules are those with effective instantiated RHS.

When a rule is fired, its instantiated RHS is actually applied to the database and returns a new database state. Therefore, there are two halting conditions to the execution of a rule. One is to test the emptiness of the matching set, the other one is to test the effectiveness of the RHS.

In a rule program, the set of all firable rules at a given time is called the *conflict set*. The semantics assigned to a rule program is captured by the iterative procedure that consists in: (i) computing the conflict set, (ii) choosing one rule among the conflict set, and (iii) applying it to the database, until the conflict set becomes empty.

2.4 Examples of rule programs

In this section we give some examples of rule program. An RDL1 rule program is decomposed into rule modules. A rule module admits a set of relations as input (called source relations), and a set of relations as an output (called target relations). A rule module is the smallest compilation unit of the language. It can be read, and eventually modified without affecting the compilation of the rule base. A module is thus seen as a "black box" whose accessible entries are the target relations. These relations can in turn be used to build new modules.

. Recursion over sets; Multiple updates.

Let PARENT be a base relation where the domain of Children is a set of names: *setOfChildren*.

Father	Mother	Children
Peter	Mary	{Louis, John}
Philippe	Louise	{Mary, Anne}

A rule module which solves the ancestor problem for this PARENT relation is the following one:

```
MODULE ANCESTOR ;
target ANCESTOR(Father:text, Mother:text,
                Desc : setOfChildren);

PARENT(x) →+ ANCESTOR(x);

PARENT(x) AND ANCESTOR(y) AND ((member
      (y.Desc, x.mother) = true) OR (member
      (y.Desc, x.Father) = true))

      → - ANCESTOR (y) + ANCESTOR (Father =
y.Father,Mother=y.Mother,Desc=union(y.Desc,x.
Children));

end.
```

In this example, the two UDF **member** and **union** are defined for the *setOfChildren* domain. The **member** operator takes two parameters : the first one is the name of the parent that is to be identified in the second parameter which is the set of children. The **union** operator builds the set of children based on the union of both sets passed as parameters. After the firing of the previous module, the instance of the ANCESTOR relation is as follows :

Father	Mother	Desc
Peter	Mary	{Louis, John}
Philippe	Louise	{Mary,Anne, Louis, John}

. Negation vs deletion

In certain cases [Abiteboul89], negation in LHS of rules is equivalent to deletion in RHS. This is the case for the following simple program.

Base relations are PENGUINS, CROWS having for schema {Name, Country, Predator}. The WINGS_DEL module computes the animals which fly. The predicate DONE avoids the rule program to loop over the third and fourth rules.

```
MODULE WINGS_DEL ;
target FLY (Name:text) ;
PENGUINS (x) →+ BIRDS (Name = x.Name) ;
CROWS (x) →+ BIRDS (Name = x.Name) ;
BIRDS (x) AND not DONE(x)→ + FLY (Name = x.Na
PENGUINS (x) → - FLY (Name = x.Name) ;
end.
```

The next module is equivalent to the previous one and uses negation in LHS of the rule.

```
MODULE : WINGS-NEG ;
target : FLY (Name) ;
rules :
PENGUINS (x) →+ BIRDS (Name = x.Name) ;
CROWS (x) →+ BIRDS (Name = x.Name) ;
BIRDS (x) AND NOT PENGUINS (x)
    → + FLY (Name = x.Name) ;
end.
```

3 AN EXECUTION MODEL FOR RULE PROGRAMS

3.1 Motivations

One of the main problems encountered in the framework of Deductive Database is how to evaluate a rule program, that is a query. Most query processing algorithms for deductive databases rely strongly on graph models. Such models have been proposed in the literature for a logic based language such as DATALOG, (see for instance [Aly87] for a Petri-Net based model). In order to model the update capabilities of RDL1 and to obtain a tight integration between the deductive capabilities of the rule language and the DBMS, we have proposed [Maindreville88b] another Petri-Net based model, called Production Compilation Network (PCN). In the next section, we present through some examples, the PCN model used to implement the RDL1 language.

3.2 The Production Compilation Network

The structural aspect of a PCN represents the relationships between rules and relational predicates as specified by a rule program. The following associations can be made between rules and the PCN structure. We represent each rule by a transition and each relational predicate involved in a rule by a place. The relational predicates that occur in the LHS of a rule are input places to the transition representing the rule, and the relational predicates that occur in the RHS of the rule are the output places of this transition. The condition of a rule is represented in the transition inscription.

The dynamic aspect of the PCN consists in the semantics associated to the firing of the PCN transitions. This corresponds precisely to the semantics of the RDL1 language, i.e PCN can model the behavior of any RDL1 program [Maindreville88b]. In the following we give examples of PCN .

The PCN of Figure 3.1 corresponds to the ANCESTOR Module defined in the previous section. Two relational predicate names appear in the rules. They lead to the places PARENT and ANCESTOR (represented by the circles P and A on the net). The first rule is modelled by the transition T1 whose inscription is the formula TRUE. The second rule is modelled by the transition T2, whose inscription is F: **member**(y.Desc, x.mother) OR **member**(y.Desc, x.father). On the net, transitions are represented by boxes. The arcs outgoing from PARENT are labelled by x. The arc outgoing from ANCESTOR is labelled by y. Finally, the arc going from T2 to ANCESTOR is labelled by -y + (y.Father,y.Mother, **union**(x.children,y.Desc))

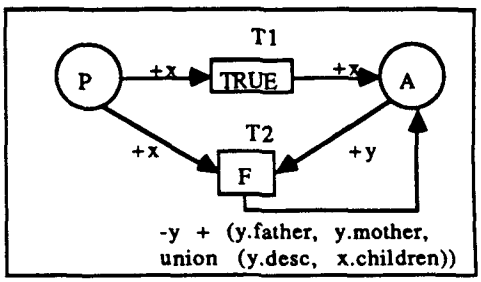


Figure 3. 1 : *PCN for the Ancestor rules*

An SQL query is represented as a single transition with several input places and a single output place representing the result of the query. Thus, a deductive query can be represented as the combination of two PCNs. The first one represents all the rules needed to define the content of the input places of the query. The second net represents the query itself. The resulting PCN is called a *query PCN*.

A PCN is a uniform execution model for implementing updates and triggers in a deductive database. The PCN model provides (i) a good indexing scheme of rules storage structure, and (ii) a formal framework to describe general computation strategies and query optimization algorithms [Maindreville88b].

4 THE ARCHITECTURE OF THE RDL1 SYSTEM.

In this section, an overview of the functional architecture of the RDL1 deductive database system is given. Then, the specific modules devoted to the compilation, optimization and execution of rules are described in detail.

4.1. Process structure and functional architecture

The general system architecture is based on the architecture of the SABRE relational database system [Gardarin86]. It is divided into three functional layers: i) the language parser, ii) the rule evaluator and the relational query optimizer, and iii) the algebraic and physical machine. The functional architecture of the system is portrayed on the figure 4.1. The shaded part represents the traditional code of a relational system. The white parts essentially represent the additional code to obtain a deductive database system.

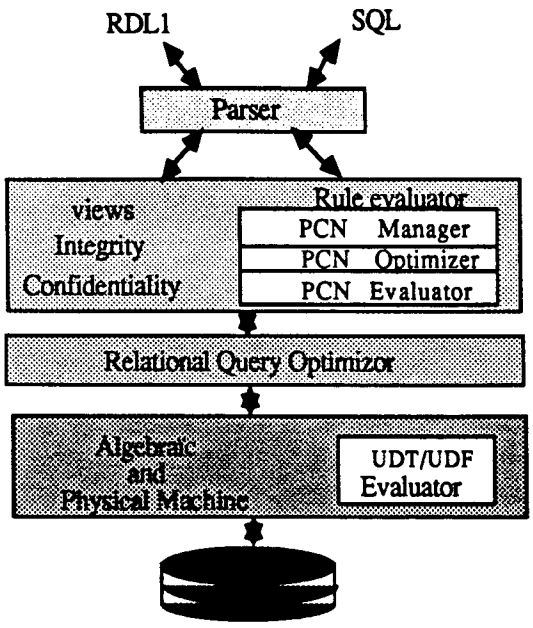


Figure 4.1 : Functional architecture of the RDL1 system

4.1.1 The language parser

In the functional system architecture, the two main interfaces of the deductive database are the rule language and an extended version of SQL. The RDL1 parser compiles the rules into an internal representation called PCN. This PCN is then stored in the relational database.

SQL is used to query derived or base relations (RDL1 is fully compatible with SQL), and also to define integrity constraints and user defined data types and operators.

4.1.2 The rule language evaluator

When a query involving a derived relation is submitted to the system, the first step consists in retrieving the pertinent PCN, i.e, the minimal PCN which is able to produce facts in the deduced relations involved in the query. The optimization phase follows. This phase is divided in two subparts. The first one consists of a logical rewriting of the PCN. This is done using several transformation techniques which will are detailed in the next section. The second optimization phase is performed at run time and consists in efficiently managing the set of firable rules. Firing a rule generates a relational query which is processed by the Query Evaluation Processor (QEP). The QEP performs the decomposition of relational calculus expressions into an optimized sequence of relational operators. Thus, rule programs are solved by repetitive calls to the QEP issued by the PCN evaluator.

4.1.3 The algebraic and physical machine

At the lowest level, the basic operations of the RDL1 language are those of relational algebra extended with aggregates and functions. These are supported by an algebraic machine (a part of the SABRE relational database system [Gardarin86]). The physical machine accesses the data on disks and managesv data sharing and reliability. To process these operations efficiently, access methods are managed, a cache memory is used, efficient algorithms for join and selection operations are implemented.

User Defined data Types and Functions appearing in RDL1 rules and in SQL queries are handled by the UDT/UDF evaluator. The filtering processor of the algebraic machine performs selections and evaluates functions over attribute values. UDF are evaluated with repeated calls to the UDT/UDF evaluator by the filter.

4.2 Rule storage and retrieval

As observed in the previous section, PCN can be used to model interrelationships between predicates and rules. In our implementation, the PCN model is used for compiling rules, and provides the internal form for storing rule modules in the database. Our proposition for storing rules in the database using PCN offers three main advantages.

First, it allows a homogeneous management of rules and facts in a deductive database system. Thus, standard features of database systems such as physical and semantic integrity, recovery and sharing can be applied to the management of the rules. Moreover, rules can include predicates ranging over the relations that store the rules. Meta-knowledge can thus be easily integrated and handled with rule programs.

Second, it allows rules to be stored in a pre-compiled form using the descriptive power of the PCN model;

Third, a rule base is stored in a compact form which is easily maintained. This last property permits efficient access to the relevant rules when a query has to be processed.

The schema of the rule database consists of five relations described below.

`MODULE (#module, module_name)` is a relation identifying the rule modules with an id and a user given name

`PERTMODULE(#module, #relevant_module)` is a relation defining for each module the relevant modules that participate in its definition.

`TRANSITION (#trans, predicate, recursive_transition, explanation)` is a relation describing each transition with a transition id, the associated predicate, whether it is recursive or not, and a textual explanation of the rule.

P_TRANSITION(#module, #place, edge_label, #transition) is a relation describing edges that link a place to a transition.

TRANSITION_P(#module, #transition, #place, edge_label) is a relation describing edges that link a transition to a place.

A more detailed discussion about rules storage and retrieval is available in [Cheiney89].

4.3 Optimizing and executing rule programs.

In a deductive database system, optimization and execution strategies should be understood together. In our system, rules are computed in two main phases. The first one performs a logical optimization of the query PCN. The second one performs an efficient management of the conflict set.

4.3.1 Logical rewriting of rule programs.

The goal is to rewrite a PCN into an optimized one. Such a transformation is performed in the three main steps described below.

• reduce the number of transitions appearing in a PCN . This reduces the number of calls to the Query Optimizer. We designed an algorithm which consists in traversing the PCN and, for each place P, grouping its output transitions with its input transitions. This merge operation depends on syntactic conditions over the net. This technique can be seen as an extension of query modification. Merging the LHS of several rules into a single transition makes possible a global optimization of the relational expressions contained in this transition. This point as a general optimization technique was also studied in [Sellis88, Krishnamurthy86].

• recognize common sub-formulae in different transitions in order to filter tuples that satisfy this formula once for all transitions [Maindreville87]. The same technique also applies to a recursive transition if it contains a sub-formula whose truth value does not change between two firings.

• move up the selection predicates next to the base relations. This well known technique consists in performing selection operations as soon as possible during the processing of the PCN. This problem calls for special algorithms, in particular for recursively defined relations. In the RDL1 system, we implemented an algorithm which performs such a task for recursive linear rules [Maindreville87]. It is a backward chaining algorithm which traverses the PCN and for each transition, recursive or not, pushes up the constants to the next upper level. Our algorithm has some similarities with one of the algorithm proposed in [Sellis88].

4.3.2 Firing rules using extended relational algebra programs

The execution phase is a cycle consisting of three actions: match, select, and fire. First, the interpreter finds all the relevant transitions whose conditions *match* the current database state. The PCN evaluator takes a transition and the relational query optimizer processes it. It decomposes the transition into an optimized set of relational algebra expressions which are *executed* by the algebraic and physical machine. A temporary relation that corresponds to the matching set of the transition is returned. Only those transitions whose matching set is not empty are said relevant. Among this set of transitions the *conflict set* is built with all the firable rules (i.e., the ones that can change the database state). To determine it, the PCN evaluator computes the effect of each relevant transition. This requires to process a union (or a difference) operation between a place and a projection of the matching set for each output arc of the transition. Finally, the PCN evaluator *selects* one transition in the conflict set and *fires* it. We next review the main issues of optimization in this execution cycle.

A first issue is to choose the next transitions that should be considered for relevance at any step of the execution cycle. This will determine the number of rules examined concurrently in the conflict set. A strategy for such a choice is called the conflict resolution strategy. In our system, the conflict resolution strategy is based on firing by stepwise saturation and chaining. It induces an implicit partial ordering among the transitions of the conflict set. This ordering is defined as follows : If two transitions t_1 and t_2 are in the conflict set, then $t_1 \leq t_2$ iff there is a path in the PCN going from t_1 to t_2. If $t_1 \leq t_2$ and $t_2 \leq t_1$ then $t_1 = t_2$. This means that the content of a relation must reach a stable state before considering an other rule using it. In particular, this strategy implements the notion of *stratification* defined for logic programs. From an optimization point of view, this strategy minimizes the number of relational queries used to execute the PCN. For instance, it leads to fire a non recursive rule only once. Also, this strategy implies that there is no more than one rule in the conflict set at any time. Indeed, a rule is first tested for relevance according to the partial order. If the test fails, a next rule is chosen in the hierarchy. Similarly, if a rule is relevant, one tests if it is firable. If not, an other rule is chosen.

Firing a transition implies recomputing the conflict set because the database state has changed. This task is the major time consuming part of the execution cycle. Thus, it deserves special effort for query optimization. Two main tasks are involved in the recomputation. One is to recompute the matching set of a rule, and second is to compute the effect of the rule. Such a task can be

time consuming when considering non monotonic programs because (i) some rules for which the test of relevance failed at one step may become firable, and (ii) some rules that were fired may become firable again. This means that a similar task can be repeated on consecutive cycles. We proposed a first algorithm [Regnier89] which maintains some information accross the execution cycles , in order to optimize this task.

The last issue is the processing of a rule at the level of the algebraic and physical machine. The problem is to efficiently store temporary relations in main memory in order to perform (i) fast evaluation of the matching set of a transition, and (ii) fast unions and differences involved in the firing of a transition. Standard relational DBMS do not treat temporary relations as first class citizens. Indeed, most of the DBMS store temporary relations as sequential files. The first measurements we performed on our prototype exhibit that improvements could be obtained by using indexing schemes for temporary relations. Nevertheless, assuming that temporary relations can be indexed in main memory, determining the most appropriate way of indexing them is a very complex task. The PCN model needs to be extended to capture physical information such as sorted relations, cardinality of relations, indexing schemes. A first step in this direction is knowing which temporary relations have already been sorted. A better integration between the PCN evaluator and the relational query optimizer should enhance such improvements.

5 IMPLEMENTATION

In this section, we describe the implementation details of the rule evaluator and the user-defined data types and functions. The approach to production rules is an integrated one whereby the rule program evaluator is constructed within the database server and uses internal database operations to perform PCN evaluation. The UDT/UDF evaluator is at the core of the DBMS system and is called by the filter.

5.1 Implementation of an extendible rule evaluator.

This integrated approach to production rules is performance sensitive. Rule programs are compiled into an internal representation which is more efficient to manage than the external language syntax. Low-level operations, not available at the external language level (SQL) are used appropriately. An ad-hoc interactive programming environment is provided to implement the rule evaluator. In this environment, two kinds of basic primitive operations are made available to the programmer. First are the DBMS internal functions for passing relational queries to the query optimizer, and for managing temporary relations. Second are the functions that manipulate the structure of a PCN (e.g., to access to its components, to modify them). The programmer enters this environment via a particular command of the extended SQL language. There, he may either consult all the functions that make the rule evaluator or modify them, or add new functions. The registration of new functions into the RDL1 system code is done using specific commands. It does not require a recompilation of the entire system code.

The main advantage of such a programming environment is to offer a system extensibility through the rule evaluator. This concept of extensibility is close to the one investigated for database programming [Carey86, Batory88]. Indeed, there are different strategies for evaluating rule programs. Some may be more efficient than others when considering assumptions about applications. Also, slightly different semantics can be assigned to rule programs [Brownston85, Abiteboul89]. Features like backtracking may also be useful for applications with large rule bases. We next detail how this environment is implemented.

To implement the rule evaluator in an extendible way, we used a special implementation of a LISP interpretor that was designed and integrated with the DBMS [Gardarin89]. This is the UDT/UDF evaluator. The same component is also responsible of evaluating UDF occuring in relational queries. Therefore, the rule evaluator is a LISP program which evaluates the PCN until it has reached a stable state. A set of primitive operations correspond to basic operations over the PCN and its elements which are places, arc labels, transitions and relations. For example, relations are accessible as LISP objects. At the PCN evaluator level, operations performed over relations are UNION, INTERSECTION, MERGE, DIFFERENCE, REMOVE-DOUBLES, TUPLE-COUNT. Calls to these LISP primitive operations are translated into direct calls to the database operators. Results of the operations are also returned as LISP objects.

User defined functions, which implement the rule evaluator are stored in the (meta)database. Different evaluation strategies can thus be implemented and stored in the database. The UDT/UDF interpreter is aware of the (meta-database) relation that stores user-defined functions, and automatically retrieves functions when they are needed.

The external interface to the rule evaluator is the following :

```
procedure rule_eval (program_name : text,
            PCN : S-expression, query : Q-tree);
begin
      result := APPLY (program_name, PCN) ;
      QUERY.ANSWER := result
end ;
```

The rule_eval procedure accepts three parameters. The first parameter is the name of the LISP program that is to be used to evaluate the PCN. Then the PCN follows. It is a LISP S-expression. The QUERY parameter contains the query and the answer field which is used to store the rule program result. The apply function is the LISP apply function which starts the execution of the rule evaluation program against the PCN.

5.2 Implementation of the UDT/UDF evaluator.

The implementation of the LISP interpretor described for PCN evaluation is also at the heart of UDT/UDF evaluation. The UDT/UDF evaluator is called by the filter to evaluate functions used in relational expressions. For each selected tuple, the UDT/UDF evaluator is called with the function name and tuple values selected as function parameters. It returns one result per tuple. As for the rule evaluator, the interpreter is aware of the database relation used to store user-defined function code. Referenced functions which are not in memory are searched for in the database.

Seeing that the interface to the UDT/UDF evaluator is contained in the most inner loop of the DBMS program, design choices for the interpreter were made to respect performances.

In this implementation of LISP, optimizations could be made because the UDT/UDF evaluator is not an end-user programming environment where environment variables need to be maintained. Indeed, only function code persists between evaluation cycles. It can be identified as such when read into the interpreter. All other memory can be reused between cycles. This strategy eliminates the costly problem of garbage collection whereby the entire memory needs to be traversed or reference counts managed. Furthermore, tuple values are considered as non-symbolic. This optimizes the READ phase because no symbol dictionary needs to be managed. Also, simple values such as numbers and character strings are returned in their binary representation as function results.

The interface to the UDT/UDF evaluator is the following:

```
Procedure ApplyUdt (opname:text;
      arglist:list):result;
 begin
  lispargs := convert (arglist) ;
  ApplyUdt := Apply(opname, arglist) ;
end;
```

This simplified procedure receives two parameters which are respectively, the name of the UDF to apply and the list of values to which the function is to be applied. The argument list is converted into a representation manageable by LISP. Then, the function is applied to this list using the standard LISP Apply function which returns the result of the operation.

6 CONCLUSIONS

This paper presented an overview of the RDL1 system. It is a rule based programming environment which is integrated with an extended relational DBMS supporting user defined data types. The RDL1 language is then an extension of the Datalog language towards the support of negation, updates, and functions. RDL1 programs are compiled into an internal execution model called Production Compilation Network (PCN). This internal representation is easily managed by a relational database. A set of rules forms a connected graph, the PCN, which represents the rule program. The PCN can be altered by the PCN optimizer before running the query evaluator. The PCN evaluator traverses the PCN to execute the rule program over the database.

Conclusions derived from first performance results of the RDL1 system follow. Other measures are currently being performed.

In the RDL1 system, the support of operations over complex domains and the support of complex updates are the most time consuming part of the computation process. For updates, the extensive use of union and differences requires to optimize these operations. An interesting discussion of this problem can be found in [Sellis85], where rewriting techniques are provided to replace a sequence of append and delete commands into optimized ones. At the execution level, most of the time is spent in materializing temporary relations, and checking the termination of rule firing (which involves the difference operation). In order to be more efficient, we envision implementing the rule language over new data structures based on inverted graphs which precompile relational operations providing direct access to tuples in main memory [Pucheral89].

A good storage structure for the rule programs has to be provided. Two points have to be taken into account : first, the rule storage structure must permit an efficient extraction of relevant rules. In order to be efficient, we designed a special algorithm based on a double hashing of the join attributes [Cheiney89]. Secondly, the rule storage structure must permit to precompile useful information for the query processing phase such as rule recursivity, access methods defined on relations...

The operations computed over complex domains, UDT can also be implemented in C and compiled into object code. The interpreter dynamically loads user-defined C programs. LISP and C programs can be combined. For example the definition of the SURFACE operator for polygons can be implemented in C and registered. Errors arising in C programs may cause the database process to

terminate abnormally. Another point we have studied [Cheiney88] is the possibility to cluster relations according to the result of user-defined operators applied on basic or user-defined domains.

The RDL1 system was developed and prototyped at INRIA on UNIX workstations. The current prototype implements all the functionalities presented in this paper and is fully integrated in the relational DBMS. It has been demonstrated at the ACM SIGMOD'89 Conference.

Acknowledgements: We would like to thank M. Jean-Noel, N. Lefebvre, D. Pastre, N. de Sahb and H. Stora for their participation in the implementation effort of the RDL1 system.

REFERENCES :

[Abiteboul89] Abiteboul S., Simon E.: "*Fundamental properties of deterministic and nondeterministic extensions of Datalog* ". INRIA Research Report, April 1989.

[Aly87] H. Aly , Z. M. Ozsoyoglu : "*Non-deterministic Modelling of Logical Queries in Deductive Databases* ", Proc of ACM-SIGMOD, San-Francisco, 1987.

[Batory88] D.S. Batory: "Concepts for a Database System Synthesizer", proc. of PODS, Austin, 88

[Brownston85] L. Brownston, R. Farrell, E. Kant, N. Martin :"*Programming Expert Systems in ` OPS5 : An Introduction to Rule-Based Programming*". Ed. Addison-Wesley.

[Carey86] M. Carey, D. Dewitt "The Architecture of the Exodus Extendible DBMS", Proc. of the International Workshop on Object Oriented Database Systems, Pacific Grove, Sept 86

[Cheiney88] Cheiney J.P. and Kiernan G.," *A Functional Clustering Method for Optimal Access to Complex Domains in a Relational DBMS*", Proc. of Data Engineering, Los Angeles, 1988

[Cheiney89] Cheiney J.P., C. de Maindreville.: " *Relational Storage and Efficient Retrieval of Rules in a Deductive DBMS*" , 5th IEEE Int. Conf on Data Engineering, Los Angeles, Feb 89.

[Delcambre88] Delcambre, L.M.L, Etheredge J.N,: *"The relational production language : a production language for relational database"*. Proc of Int. Conf. on Expert Database System, April 1988.

[Gardarin86] Gardarin G., Abiteboul S., Scholl M., Simon E.:*"Towards DBMS's for Supporting New Applications"*, Proc. of 12th VLDB, Kyoto, 1986.

[Gardarin89] Gardarin G. et al.: "*Managing Complex objects in an Extendible Relational DBMS*", in Proc. of 15th VLDB, Amsterdam 1989.

[Genrich87] H. J. Genrich: "*Predicate / Transition Nets* " , in Advances in Petri Nets' 86. Springer Verlag, 1987.

[Krishnamurthy86] R. Krishnamurthy, H. Boral, C. Zaniolo : "*Optimization of Nonrecursive Queries*", Proc of 12th VLDB, Kyoto 1986.

[Maindreville87] C. de Maindreville, E. Simon : "*A Predicate Transition Net for evaluating Queries against rules in a DBMS.*" INRIA Research Report N°604, Feb. 87.

[Maindreville88] C. de Maindreville, E. Simon : "*A Production Rule Based Approach To Deductive Databases*", Proc of 4th International Conference on Data Engineering, Los Angeles, Feb 88.

[Maindreville88b] C. de Maindreville, E. Simon : "*Modelling queries and updates in a deductive database*" Proc of 14th VLDB Conference, Los Angeles Sept. 1988.

[Ong84] Ong J. and al., "*Implementation of Data Abstraction in the Relational Database System INGRES*", ACM-SIGMOD, rec 14, 1984, pp 1-14

[Pucheral89] P. Pucheral, J.M Thevenin : " *A graph based data structure for efficient implementation of main memory DBMS's*". 6th IWDM Conference, Deauville, France,1989.

[Regnier89] Regnier M., Simon E.: "*Efficient evaluation of Production Rules in a DBMS*", in Advances in Databases, Geneva, Sept. 1989.

[Sellis85] T.K Sellis, L. Shapiro : "*Optimization of extended database query languages*", ACM SIGMOD 1985.

[Sellis88] T.K. Sellis : "*Multiple-Query Optimization*" ACM TODS, Vol 13, No.1, March 1988.

[Simon88] Simon E., de Maindreville C.: "*Deciding whether a production rule is relational computable*" Proc of International Conference on Database Theory, Bruges, Belgium Sept 88.

[Stonebraker83] Stonebraker M. and al., *"Application of Abstract Data Types and Abstract Indicies to CAD databases"*, Proc. of Engineering Design Applications of ACM-IEEE Database Week, San Jose, Ca., May 1983

[Stonebraker86] Stonebraker M., *"Inclusion of New Types in Relational Database Systems"*, ACM-IEEE, 1986, pp 262-269.

[Wilms88] P.F Wilms et al : "*Incorporating data types in an extendible database architecture*", IBM Research report, RJ 6405, Aug 1988.