IMPLEMENTATION OF RULES IN RELATIONAL DATA BASE SYSTEMS

by
Michael Stonebraker, John Woodfill and Erika Andersen

Dept of Electrical Engineering and Computer Science
University of California
Berkeley, Ca.

ABSTRACT

This paper contains a proposed implementation of a rules system in a relational data base system. Such a rules system can provide data base services including integrity control, protection, alerters, triggers, and view processing. Moreoever, it can be used for user specified rules. The proposed implementation makes efficient use of an abstract data type facility by introducing new data types which assist with rule specification and enforcement.

I INTRODUCTION

Rules systems have been used extensively in Artificial Intelligence applications and are a central theme in most expert systems such as Mycin [SHOR76] and Prospector [DUDA78]. In this environment knowledge is represented as rules, typically in a first order logic representation. Hence, the data base for an expert system consists of a collection of logic formulas. The role of the data manager is to discover what rules are applicable at a given time and then to apply them. Stated differently, the data manager is largely an inference engine.

On the other hand, data base management systems have tended to represent all knowledge as pure data. The data manager is largely a collection of heuristic search procedures for finding qualifying data. Representation of first order logic statements and inference on data in the data base are rarely attempted in production data base management systems.

The purpose of this paper is to make a modest step in the direction of supporting logic statements in a data base management system. One could make this step by simply adding an inference engine to a general purpose DBMS. However, this would entail a large amount of code with no practical interaction with the current search code of a data base system. As a result, the DBMS would get much larger and would contain two essentially non overlapping subsystems. On the other hand, we strive for an implementation which integrates rules into DBMS facilities so that current search logic can be employed to control the activation of rules.

The rules system that we plan to implement is a variant of the proposal in [STON82], which was capable of expressing integrity constraints, views and protection as well as simple triggers and alarms for the relational DBMS INGRES [STON76]. Rules are of the form:

```
on   condition
then action
```

The conditions which were specified include:

the type of command being executed (e.g. replace, append)

the relation affected (e.g. employee, dept)
the user issuing the command
the time of day
the day of week
the fields being updated (e.g. salary)
the fields specified in the qualification
the qualification present in the user command

The actions which we proposed included:

sending a message to a user
aborting the command
executing the command
modifying the command by adding qualification or
changing the relation names or field names

Unfortunately, these conditions and actions often affect the command which the user submitted. As such, they appear to require code that manipulates the syntax and semantics of relational commands. This string processing code appears to be complex and has little function in common with other data base facilities. In this paper we make use of two novel constructs which make implementing rules a modest undertaking. These are:

1) the notion of executing the data
and
2) a sequence of QUEL commands as a data type for a relational data base system

The remainder of this paper is organized as follows. In Section II we indicate the new data types which must be implemented and the operations required for them. Then in Section III we discuss the structural extensions to a relational data base system that will support rules execution. Lastly, Section IV and V contains some examples and our conclusions.

II RULES AS ABSTRACT DATA TYPES

Using current INGRES facilities [FOGG82, ONG82, STON82a] new data types for columns of a relation can be defined and operators on these new types specified. We use this facility to define several new types of columns and their associated operators in this section.

The first data type is a QUEL command, e.g.

```
range of e is employee
replace e(salary = 1.1*e.salary) where e.name = "John"
```

The abstract data type facility supports an external representation such as that above for a given data type. Moreover, when an object of the given type is stored in the data base it is converted to an internal representation. QUEL commands are converted by the INGRES parser to a parse tree representation such as the one noted in Figure 1 for the qualification "where 13. + employee.salary = 100". Consequently, a natural internal form for an object of type QUEL is a parse tree. Each node in this parse tree contains a value (e.g. 13.) and a type (e.g. floating point constant).

The second new data type which will be useful is an ATTRIBUTE-FUNCTION. This is a notion in the QUEL grammar and stands for anything that can be evaluated to a constant or the name of a column. Examples of attribute functions include:

13.

```
                 RETBOOL
                    |
          +------F4EQ--------------+
          |                        |
   +--F4PLUS--+                  100.
   |          |
  13.       F4VAR
              |
              1
```

The Parse Tree for the Qualification
Where 13. + employee.salary = 100

Figure 1

1.1*employee.salary +20

newsal

The external representation is the same string format used for objects of type QUEL; the internal representation is that of a parse tree.

Two other data types of lesser significance are also needed, a TIME data type to contain a time of day value and a COMMAND data type to contain a value which is one of the QUEL commands.

Current built-in INGRES operators (e.g. *, /, +, etc.) must be extended for use with attribute functions. In addition, two new operators are also required. First, we need a function new() which will operate with integer data types. When called, it will return a new unique identifier which has not been previously used. Second, we require a partial match operator, ^, which will operate on a variety of data types and provide either equality match or match the value "*".

III INGRES CHANGES

We expect to create two rules relation, RULES1 and RULES2, with the following fields:

```
create RULES1(
     rule-id = i4,
     user-id = c10,
     time    = time,
     command = command,
     relation = c12,
     terminal = c2,
     action = quel)

create RULES2 (
     rule-id = i4,
     type = c10,
     att-fn1 = attribute-function
     operator = c5,
     att-fn2 = attribute-function)
```

For example, we might wish a rule that would add a record to an audit trail whenever the user "Mike" updated the employee relation. This requires a row in RULES1 specified as follows:

```
append to RULES1(
     rule-id  = new(),
     user-id  = "Mike",
     command  = "replace",
     relation = "employee",
     action   = QUEL command to perform audit)
```

If additionally we wished to perform the audit action only when Mike updated the employee relation with a command containing the clause "where employee.name = "Fred"" we would add an additional tuple to RULES2 as follows:

```
append to RULES2(
     rule-id  = the one assigned in RULES1
     type     = "where"
     att-fn1  = "employee.name"
     operator = "="
     att.fn2  = "Fred")
```

We also require the possibility of executing data in the data base. We propose the following syntax:

```
range of r is relation
execute (r.field) where r.qualification
```

In this case the value of r.field must be an executable QUEL command and thereby of data type QUEL. To execute the rule that was just appended to R1 we could type:

```
range of r is R1
execute (r.action) where r.user-id = "Mike" and
                  r.command = "replace" and
                  r.relation = "employee"
```

When a QUEL command is entered by a user, it is parsed into an internal parse tree format and stored in a temporary data structure. We expect to change that data structure to be the following two main memory relations:

```
create QUERY1(
     user-id = c10,
     command = command,
     relation = c12,
     time     = time,
     terminal= c2)

create QUERY2(
     clause-id = i4,
     type      = c10,
     att-fn1   = attribute-function,
     operator  = c5,
     att-fn2   = attribute-function)
```

If the user types the query:

```
range of e is employee
retrieve (e.salary)
      where (e.name = "Mike" or e.name = "Sally")
        and e.salary > 30000
```

then INGRES will build QUERY1 to contain a single tuple with values:

QUERY1				
user-id	command	relation	time	terminal
current-user	retrieve	employee	current-time	current-terminal

QUERY2 will have four tuples as follows:

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-x	target-list	employee.salary	=	employee.salary
id-y	where	employee.name	=	Mike
id-y	where	employee.name	=	Sally
id-z	where	employee.salary	>	30000

Notice that QUERY1 and QUERY2 contain a relational representation of the parse tree corresponding to the incoming query from the user. The where clause of the query is stored in conjunctive normal form, so that atomic formulae which are part of a disjunction have the same clause-id, while the atomic formulae and disjunctions in the conjunction have different clause-ids.

Then we execute the QUEL commands in Figure 2 to identify and execute the rules which are appropriate to the incoming command. These commands are performed by the normal INGRES search logic. Activating the rules system simply means running these commands prior to executing the user submitted command. After running the commands of Figure 2, the query is converted back to a parse tree representation and executed. Notice that the action part of a rule can update QUERY1 and QUERY2; hence modification of the user command is easily accomplished. The examples in the next section illustrate several uses for this feature:

```
range of r1 is RULES1
range of r2 is RULES2
range of q1 is QUERY1
range of q2 is QUERY2
retrieve into TEMP(r1.id, r1.quel) where
        r1.user-id ^ q1.user-id and
        r1.command ^ q1.command and
        r1.time    ^ q1.time    and
        r1.terminal^ q1.terminal

range of t is TEMP
execute (t.quel) where t.id < 0 or
      (t.id = r2.rule-id and
      set(r2.all-but-rule-id by r2.rule-id)
    = set(r2.all-but-clause-id by r2.rule-id
     where r2.all-but-rule-id ^ q2.all-but-clause-id))
```

Rule Activation in QUEL
Figure 2.

The set functions are as defined in [HELD75]. The conditions for activating a rule are:

(i) its tuple in RULES1 matches the tuple in QUERY1

and either
(ii) each tuple for the rule in RULES2 matches a tuple in QUERY2.
or
(iii) there are no required matches in RULES2
(represented by rule-id < 0).

The second condition provides appropriate rule activation when both the user query and the rule do not contain the boolean operator OR. However, a rule which should be activated when two clauses A and B are true will have two tuples in RULES2. This rule will be activated by a user query containing clauses matching A and B connected by any boolean operator. Under study is a more sophisticated activation system which will avoid this drawback.

The commands in Figure 2 cannot be executed directly because set functions have never been implemented in INGRES. Hence, we turn now to a proposed implementation of these functions.

Suppose we define a new operator "|" to be bitwise OR, and "bitor()" to be an aggregate function which bitwise ORs all qualifying fields. Then if we add the attribute "mask" to RULES2, and give each tuple for a particular rule a unique bit, the following query is correct:

```
range of t is TEMP
execute (t.quel) where t.id < 0 or
        (t.id = r2.rule-id and
      bitor(r2.mask by r2.rule-id)
      = bitor(r2.mask by r2.rule-id
         where r2.all-but-rule-id ^ q2.all-but-clause-id))
```

This solution will be quite slow, since the test for each rule involves processing a complicated aggregate. A more efficient solution involves generating masks for all rules in parallel and writing special search code as follows:

```
range of r1 is RULES1
range of r2 is RULES2
range of q1 is QUERY1
range of q2 is QUERY2
retrieve into TEMP(r1.id, r1.quel, mask = 0)  where
          r1.user-id ^ q1.user-id and
          r1.command ^ q1.command and
          r1.time    ^ q1.time    and
          r1.terminal^ q1.terminal

range of t is TEMP

foreach q2 do begin
   replace t (mask = t.mask | r2.mask)
           where t.id = r2.rule-id and
           r2.all-but-rule-id ^ q2.all-but-clause-id
end foreach

execute (t.quel) where t.id < 0 or
             (t.id = r2.rule-id and
          bitor(r2.mask by r2.rule-id)
              = t.mask)
```

Since the value of "bitor(r2.mask by r2.ruleid)" remains constant, the performance of this algorithm can be further improved by including the value of "bitor(r2.mask by r2.ruleid)" in RULES1 and copying it into TEMP as the "acceptmask". The third query would then become:

```
execute (t.quel) where t.id = r2.rule-id and
                 t.acceptmask = t.mask
```

Notice the case where there are no tuples in RULES2 for a particular rule is handled with an acceptmask of zero.

Either a variable length abstract data type "bitstring" or a four byte integer can be used to store the mask. The abstract data type solution has the advantage of allowing an unlimited number of conditions for specifying rule activation, while the four byte integer solution has the advantage of simplicity and speed, but can only represent 32 conditions.

IV EXAMPLES

We give a few examples of the utility of the above constructs in this section. First, we can store a command in the data base as follows:

```
append to storedqueries (id = 6,
             quel = "range of e is employee
                   retrieve (e.salary)
                   where e.name = "John"")
```

We can execute the stored command by:

```
range of s is storedqueries
execute (s.quel) where s.id = 6
```

The following two examples will pertain to the query:

```
range of e is employee
replace e(salary = salary*1.5) where e.name = "Erika"
```

To represent this query INGRES will append the following tuples to the QUERY1 and QUERY2 relations:

QUERY1				
user-id	command	relation	time	terminal
current-user	replace	employee	current-time	current-terminal

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-z	target-list	employee.salary	=	employee.salary*1.5
id-x	where	employee.name	=	Erika

Suppose we want to implement the integrity contraint to insure that employee salaries never exceed $30,000. Using query modification [STON75] we would add the clause "and employee.salary*1.5 <= 30000". to the user's qualification with the following rule:

```
append to RULES1(
     rule-id  =  new(),  (call it id-y)
     user-id  =  *,      (matches any user-id)
     command  =  "replace",
     relation =  "employee",
     action   =  "range of Q2 is QUERY2
                 append to QUERY2(
                  clause-id = id-x,
                        type      = "where",
                        att-fn1   = Q2.att-fn2,
                        operator  = "<=",
                  att-fn2  = "30000")
            where Q2.att-fn1 = "employee.salary")"
append to RULES2(
    rule-id  = id-y,
    type     = "target-list",
    att-fn1  = "employee.salary",
    operator = "=",
    att-fn2  = *)
```

Consider a transition integrity constraint that specifies that the maximum salary increase is 20%. This means that the new salary divided by the old salary must be less than or equal to 1.2. This can be achieved by appending a single tuple to R1:

```
append to RULES1(
    rule-id  = new(),
    user-id  = *,
    command  = "replace",
    relation = "employee",
    action   = "range of Q2 is QUERY2
              append to QUERY2(
               clause-id = id-x,
                    type     = "where",
                    att-fn1  = Q2.att-fn2/Q2.att-fn1,
                    operator = "<="
                    att-fn2  = "1.2")
           where Q2.att-fn1 = "employee.salary""
```

As a last example of an integrity constraint, consider a referential constraint that a new employee must be assigned to an existing department. Such a rule would be applied, for example, to the following query:

```
append to employee (name="Chris", dept = "Toy", mgr = "Ellen")
```

The corresponding tuples in QUERY2 would look like:

QUERY2				
clause-id	type	att-fn1	operator	att-fn2
id-z	target-list	employee.name	=	Chris
id-z	target-list	employee.dept	=	Toy
id-z	target-list	employee.mgr	=	Ellen

Implementation of the constraint requires checking that the department given in the target list of the append appears in the department relation. This is accomplished with the following rule:

```
append to RULES1(
    rule-id  = new(),
    user-id  = *,
    command  = "append",
    relation = "employee",
    action   = "range of Q2 is QUERY2
              append to QUERY2(
                    clause-id = id-z,
                    type      = "where",
                    att-fn1   = "dept.name",
                    operator  = "=",
                    att-fn2   = Q2.att-fn2)
```

```
                where Q2.att-fn1 = employee.dept"
```

Lastly, protection is achieved primarily by making use of the RULE1 relation, which pertains to the query "bookkeeping" information. Suppose we wanted to ensure that no one could access the employee relation after- hours (after 5PM and before 8AM). The following tuple would be added to the R1 relation:

```
append to RULES1(
    rule-id = new(),
    user-id = *,
    time    = "17:01 - 7:59",
    command = *,
    relation = "employee",
    terminal = *,
    action  = "range of Q1 is QUERY1
           range of Q2 is QUERY2
           delete Q1
           delete Q2
```

If the query meets the conditions, the action removes the tuples in QUERY1 and QUERY2 and thereby aborts the command.

V CONCLUSIONS

This paper has presented an initial sketch of a rules system that can be embedded in a Relational DBMS. There are two potentially very powerful features to our proposal. First, it can provide a comprehensive trigger and alerter system. Real time data base applications, especially those associated with sensor data acquisition, need such a facility. Second, it provides stored DBMS commands and the possibility of parallel execution of triggered actions. In a multiprocessor environment such parallelism can be exploited.

There are also several deficiencies to the current proposal, including:

a) Rule specfication is extremely complex. This could be avoided by a language processor which accepted a friendlier syntax and translated it into the one in this paper.

b) The result of the execution of a collection of rules can depend on the order in which they are activated. This is unsettling in a relational environment.

c) Rules trigger on syntax alone. For example, if we want a rule that becomes activated whenever John's employee record is affected, we trigger on any query having "employee.name = John" in the where clause. However if the incoming query is to update all employees' salaries, this rule would not be triggered.

d) Commands with multiple range variables over the same relation, so called reflexive joins, are not correctly processed by the rules engine.

e) Aggregate functions have not yet been considered.

f) As noted earlier, boolean OR is not treated correctly.

We are attempting to resolve these difficulties with further work.

REFERENCES

[DUDA78] Duda, R. et. al., "Development of the Prospector Consultation System for Mineral Exploration," SRI International, October 1978.

[FOGG82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System, INGRES", Masters Report, EECS Dept, University of California, Berkeley, CA Sept. 1982.

[HELD75] Held, G., et. al., "INGRES: A Relational Data Base System," Proc. 1975 NCC, Anaheim, Ca., May 1975.

[ONG82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System, INGRES," Masters Report, EECS Dept, University of California, Berkeley, CA Sept. 1982.

[SHOR76] Shortliffe, E., "Computer Based Medical Consultations: MYCIN," Elsevier, New York, 1976.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., June 1975.

[STON76] Stonebraker, M. et al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON82] Stonebraker, M., et. al., "A Rules System for a Relational Data Base System," Proc. 2nd International Conference on Databases, Jerusalem, Israel, June 1982.

[STON82a] Stonebraker, M., "Extending a DBMS with Added Semantic Knowledge," Proc. NSF Workshop on Data Semantics, Intervale N.H., May 1982. (to appear in Springer-Verlag book edited by M. Brodie)

[STON83] Stonebraker, M., et. al., "Document Processing in a Relational Data Base System," ACM TOOIS, April 1983.