RBE: A Rule-By-Example Active Database System

Ye-In Chang and Fwo-Long Chen
Department of Applied Mathematics
National Sun Yat-Sen University
Kaohsiung, Taiwan, 80424
R.O.C.
(email: changyi@math.nsysu.edu.tw)

SUMMARY

In an active database system, rules are used to monitor and respond to events that happen inside the database. This paper presents the implementation of an active database system called RBE. The system loosely couples OPS5 production system and INGRES database management system to efficiently monitor databases on complex conditions. In this system, a table-based Rule-By-Example language, which is also noted as RBE, is designed. The internal representations of the RBE rule language are production rules; therefore, rules can be stored, managed, and tested efficiently by using the well-developed pattern matching algorithm in a production system. In other words, the system applies a production system and a special production system program to the task of query rewrite trigger processing. Moreover, a user-friendly interface is used to loosely couple OPS5 rule system with INGRES DBMS. The architecture used in this system shows the applicability of constructing an active database system by integrating any production system and any database system. Moreover, the proposed technique could be used as an implementation method for a query-rewrite rule system inside a DBMS server, not using a layered approach.

Key Words: active database systems, alerters, artificial intelligence, production systems, rules, triggers

Introduction

In recent years, various approaches have been suggested for adding active capabilities to database systems in order to integrate rules and facts and thus simplify the application programming task. The addition of active capability to database systems was first considered in order to support specific DBMS functions, such as view maintenance and integrity constraint enforcement [1], which motivates the design of an *active* database system. In an active database system, rules are used to monitor and respond to events that happen inside the database [2, 3]. Rules can reduce the amount of code that is required in applications that access the database. Rules can also prevent redundant code that would otherwise be required when multiple programs perform the same operations on the database. A trigger is one type of rules in an active database system, which is used to detect some conditions that happen in a database and then react to the database [4, 5]. Triggering mechanisms of different types have also been suggested to support the maintenance of materialized views, snapshots and derived attribute values. Moreover, the functions of rules can enforce accounting rules, automate departmental practices, perform calculations, enforce protection, create an audit trail, maintain the integrity of the database, support version control and so on. The essential research in an active database system is to design a user-friendly language to specify rules to monitor events that happen in a database, and to design a rule manager to store, manage and test rules efficiently.

Over the last decade, many researchers have proposed active database systems, [6, 7, 8], such as POSTGRES [9, 10, 11, 12, 13, 14], HiPAC [15, 16, 17], Ariel [18, 19] and Starburst [20, 21]. The POSTGRES system is a next-generation extensible database management system that can support abstract data types, procedure types and rule types, and business data types, for data, object and knowledge management. The HiPAC system, an active, time-constrained, object-oriented DBMS, deals with two main problems: one is handling timing constraints in databases, and the other is avoiding costly polling by using situation-action rules. The Ariel system is based on a production system model, which uses a specially designed discrimination network and a rule-action planner that takes advantage of the existing query optimizer. The Starburst system is a prototype relational database system with a focus on extensibility. The rules of the Starburst system are set-oriented,

and the rule system fully integrates rule definition and execution with database processing.

In these active database systems, they are designed from scratch; that is, they tightly couple the rule manager with the database. Although some researchers [22] argue that a loosely-coupled system is likely to perform poorly unless the application has certain specific characteristics, some other researchers [23, 24, 25] argue that a loose-coupling approach has to work due to the political and economic necessities and due to a loose-coupling approach concerning wider applicability and flexibility. Note that by loose coupling two systems, we mean that both systems will maintain their own functionality and will communicate through a well-defined interface [23, 24], for example, stream processing [25]; while tight-coupling, on the other hand, implies that at least one system has knowledge of the inner workings of its counterpart, and that special performance-enhancing access mechanisms are provided. The idea of a loose-coupling approach is similar to the idea of reusability in software engineering, which is more economic than a tight-coupling approach. Moreover, under the consideration of certain criteria such as applications, preexisting environment and organizational factors, a loose-coupling approach may be a preferred approach [23].

The key performance issue in the rule manager is the time required to identify which rule or rules to apply. Some of the rule managers in these active systems use a *value-driven* mechanism; that is, rules are tested after data has been accessed. Let's consider an example of a *salary-bonus* trigger "whenever a DBMS updates an employee's salary, the DBMS should also update the employee's bonus to the same value as this employee's salary". If 1000 employees' salaries are updated, a rule manager using a value-driven mechanism needs 1000 times of interaction with the DBMS to update the employees' bonuses. This mechanism causes fine granularity and intensive interaction between the DBMS and its rule manager. In POSTGRES, in addition to support the above mechanism to process rules, which is called the *marking* strategy, there is another rule processing strategy, called the *query rewrite* strategy. In the query rewrite strategy, a rule could be applied by converting a user's query to an alternate form prior to execution. This transformation is performed between the query language parser and the optimizer. In the example of the *salary-bonus* trigger, if 1000 employees' salaries are updated, the query rewrite strategy will derive a new query "update the employees' bonuses to the same values as their salaries" and then

send this query to the DBMS. Therefore, the strategy reduces the intensive interaction between the DBMS and the rule manager. This query rewrite strategy can be considered as an *event-driven* mechanism, in which it actives rules and derives the set of queries from these applicable rules before the user's query accesses the data in the database. The performance of this *event-driven* mechanism is good when there are a number of rules on any given constructed type and those rules cover the whole constructed type. However, in the implementation of the query rewrite strategy in POSTGRES, it requires four steps to derive a new query [12].

Actually, the *event-driven* mechanism well matches the pattern-matching algorithms used in a production system, if a pattern contain an event, i.e., the user's query. Moreover, the rule processing task becomes simple and economic in a production system, since there are many well developed pattern-matching algorithms, like RETE [26] and TREAT [27], which have been implemented in a production system, even in parallel, such as parallel OPS5 that is freely available from Carnegie Mellon University [28]. Therefore, based on the *event-driven* mechanism, in this paper, we design and implement an active database system, called RBE that denotes Rule-By-Example, which loosely couples a production system, parallel OPS5 [28, 29], as the rule manager, and a database system, INGRES, which is the only available traditional relational database in our department, to construct an active database system, instead of designing from scratch as in other active systems. In this system, a Rule-By-Example language that is also noted as RBE is designed, which has similar syntax as one of relational database languages, Query-By-Example [30]. The internal representations of the RBE rule language are production rules. A user-friendly interface that is written in C is used to loosely couple OPS5 rule system with INGRES DBMS as shown in Figure 1. This interface accepts user's query, inserts this event into OPS5 rule system to activate rules and sends the original query with these derived queries to INGRES DBMS. The system has been implemented on IBM RISC System/6000 workstations by using C, embedded SQL to call INGRES, and OPS5 languages.

Figure 1: The Structure of RBE

The Language

In this section, we present the syntax of a table-based rule language: Rule-By-Example that is noted as RBE, describe its internal representation, and show several examples written in this language.

Syntax of Rules

Based on the requirement of easy-to-use and complete semantic expressibility, a table-based rule language: RBE, to monitor a database is designed. To input a rule, a C program called RMX is written to offer users five options: *add, update, delete and inquire* rules or *quit* from the system. The input format of a rule is shown in Figure 2.

A *rule name* field is used to specify a unique rule name for a user's convenience. A *priority* field records the priority of this rule to control the order of rule execution when conflicts occur among applicable rules. An integer number between 1 and 5 can be chosen to represent the priority from low to high. A *condition table* field specifies these relations which are to be referred in the condition part, and an *action table* field specifies the relations which are to be referred in the action part. A menu-driven user-interface is provided to enter table names into these two fields. The RMX module has pre-loaded information about relation names and related attribute names by using embedded SQL [31].

A table inside the condition window can be filled in with any attribute name of the

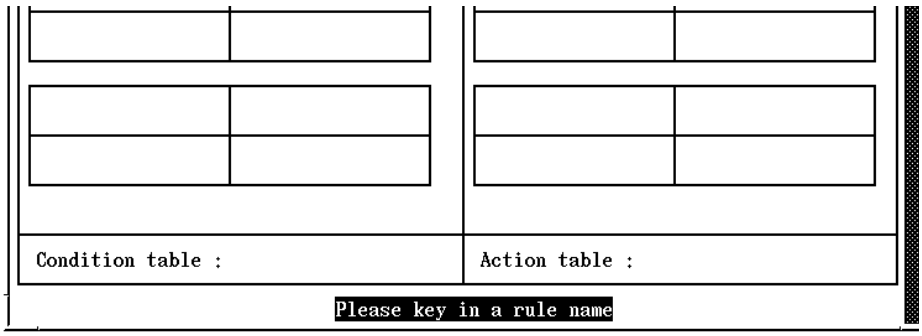


Figure 2: Rule-By-Example

relation provided by the menu. This is one more attribute *DBMS OP* provided by the system as shown in Figure 3. This attribute is used to represent an event in a DBMS, and the possible values for this attribute are *insert, delete, update* and *retrieve*. The values in the other attributes have the same expression as that in OPS5 language. For example, <x> represents a variable x which can match any value in the database. Similarly, a table inside the action window can be filled in with any attribute name of the relation. Besides, one more attribute *DBMS OP* is provided by the system. This attribute is used to represent the event which is executed by the DBMS, and the possible values for this attribute are *insert, delete, update, retrieve, refuse* and *show*. Note that the operator *show* is used to display a message on the screen and the operator *refuse* is used to reject the incoming event specified in the condition table.

When the value of *DBMS OP* is one of *insert, delete, retrieve* operations, the attributes provided by the system is the same as those in the relation. When the value of *DBMS OP* is a *show* operation, the system provides one more attribute *message*. When the value of *DBMS OP* is a *update* operation, the number of attributes provided by the system have been doubled. For each attribute A in a relation, the system provides one

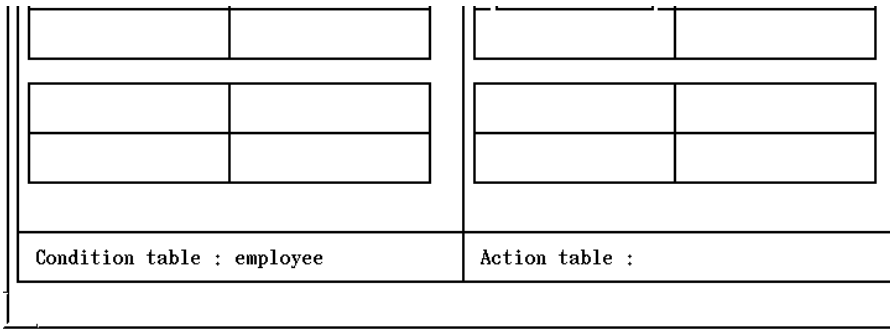


Figure 3: The *DBMS OP* field in a condition table

more attribute U_A to represent the new value after data modification as shown in Figure 4. Therefore, the old and the new values of an attribute A can be simply distinguished by fields A and U_A, respectively.

The semantics of these two windows is that if the event, the query, specified in the condition window occurs, then the DBMS should execute the event specified in the action window. Figure 5 shows an example of a trigger. The semantics of this rule is described as follows:

if Update department set mgr_no = <y> where dno = <x>
then Update employee set mgr_no = <y> where dno = <x>

That is, whenever the manager of a certain department is changed, the related manager information should also be changed in the employee table. Note that the examples shown in this paper are copied down when the system is in an Inquire Rules mode; therefore, a message "Do you want to inquire any rule?" is shown at the bottom of the screen.

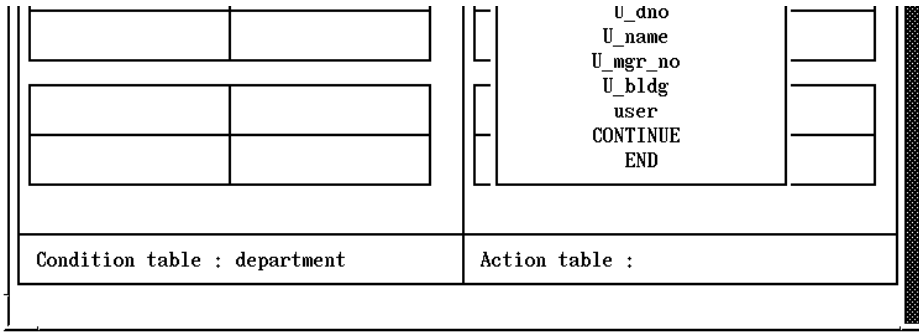


Figure 4: Attributes provided by RMX when the *DBMS OP* = update

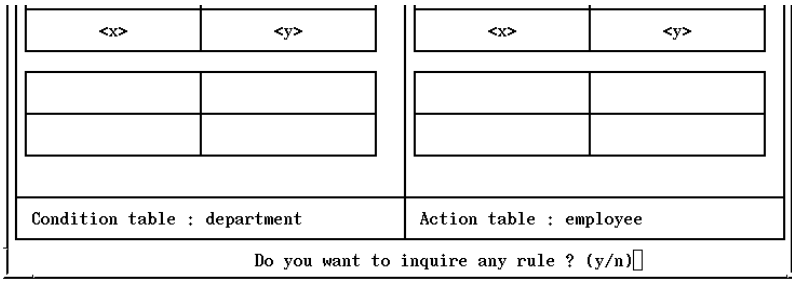


Figure 5: A trigger

The Internal Representation

For efficiency concerns in the storage, management, and tests of rules, a RBE rule is translated into an internal representation in the form of an OPS5 production rule by the RMX module.

A rule in OPS5 has the following general form:

(p <identifier>
<condition1> <condition2> ... <condition$n$>
- ->
<action1> <action2> ... <action$m$>) where n, $m >= 1$.

Each condition is a triple of object-attribute-value, and is called an *element*. For example, element (EMP ^name Mary ^bonus 40000) means that element EMP has two attributes: *name* and *bonus*, and the values of these attributes are Mary and 40000, respectively. Events in the action part can make, remove or modify an element.

Basically, a RBE rule contains two queries: one is in the condition part, and the other one is in the action part. A query is translated into a form of OPS5 elements by an *attribute-value* mapping method. For each relation referred to in the query, the relation name is used as the element name. These attributes names in such an element is the same as that in the relation, and so does the value related to each attribute. Figure 6 shows the BNF grammar of the internal representation of the RBE rule language. Therefore, the internal representation of the example shown in Figure 5 is represented as follows, where attribute DBMS OP is replaced with *dbms*:

(p trigger
(department ^dbms update ^dno <x> ^U_mgr_no <y>)
- ->
(make employee ^priority 3 ^dbms update ^dno <x> ^U_mgr_no <y>))

Examples

In addition to the example of a trigger, alerters and referential integrity are some other useful functions in an active database system. An alerter is another type of rules in an active database system, which is used to monitor a database and then report the changes

rule	::=	(p *rule_id conditions* −− > actions**)
conditions	::=	(*condition_table* ^dbms *condition_operator condition_field**)
actions	::=	(make *action_table* ^priority *number* ^dbms *action_operator* *action_field**)
condition_operator	::=	delete | update | retrieve | insert
action_operator	::=	delete | update | retrieve | insert | show | refuse
condition_field	::=	^*field_name condition*
action_field	::=	^*field_name action*
condition	::=	{*restriction**} /* conjunctions */ | *restriction*
restriction	::=	<< *any_atom** >> /* disjunctions */ | *predicate atomic_value* | *variable predicate actomic_value* | *atomic_value*
atomic_value	::=	*constant_symbolic_atom* | *number* | *variable* /* <symbol atom> */
predicate	::=	= | <> | < | <= | >= | >
action	::=	*atomic_value* | *expression*
expression	::=	*number* | *variable* | *expression operator expression* | (*expression*)
operator	::=	+ | − | ∗ | /
rule_id	::=	*constant_symbolic_atom*
condition_table	::=	*constant_symbolic_atom*
action_table	::=	*constant_symbolic_atom*
field_name	::=	*constant_symbolic_atom*

Figure 6: BNF Grammar for RBE

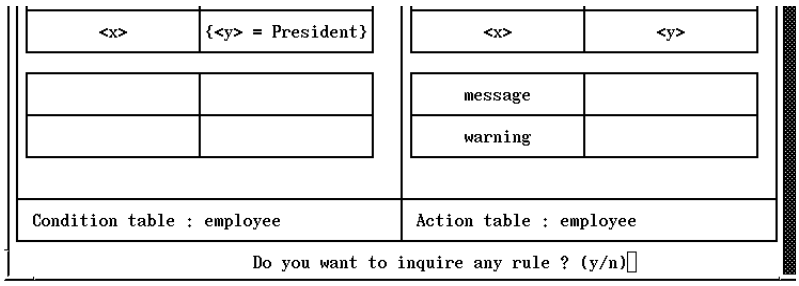


Figure 7: An alerter

to users. Figure 7 shows an example of an alerter rule in which whenever an employee's job is updated to President, the system should show a warning message. In order to maintain the consistency of a database, referential integrity is a required service. Figure 8 shows an example of referential integrity, which states that if a tuple in relation department, where department.dno = <x> is removed, then there should be no employee working in dno = <x> department.

Most of the commercial DBMSs do not allow views to be updated since it may result in data inconsistency in a database. The RBE system provides users to maintain view consistency by writing rules. Figure 9 shows an example of view consistency control. The view *emp* is defined over employee and department tables. Whenever the values in the view are updated, related values are updated.

In the above examples, all the rules are event-driven. That is, rules are activated before the events are executed by the DBMS. There are still some rules which are activated only after the events are executed by the DBMS, i.e., value-driven rules. For value-driven rules, the system most pre-load some related data. Figure 10 shows an example of a value-driven rule since the rule can be activated only when the value of the job attribute is known. This rule states that the president's salary cannot be known. That is, whenever a query retrieves the president's salary, this event should be refused. For our system to work

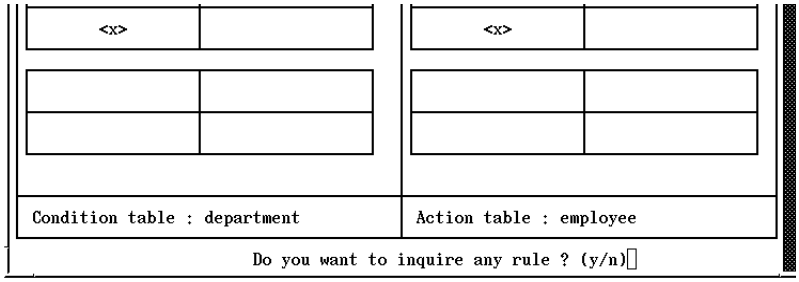


Figure 8: Referential integrity

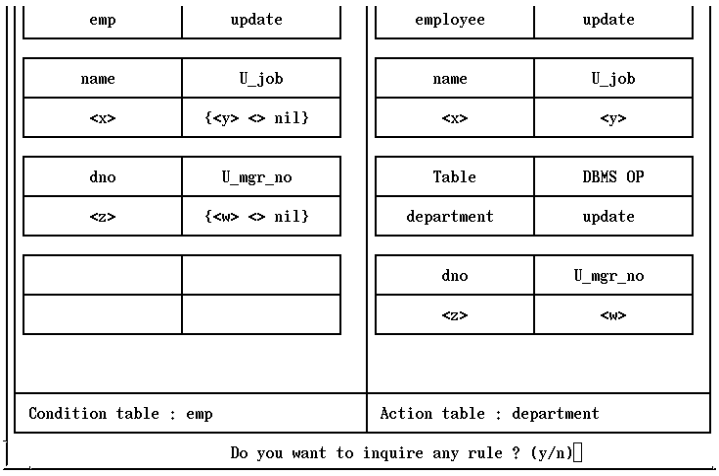


Figure 9: View consistency control

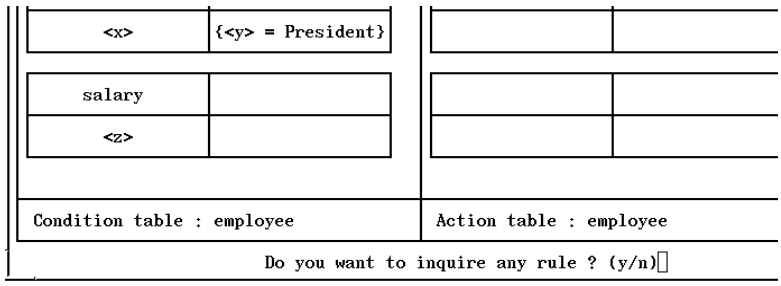


Figure 10: A value-driven rule

well in this case, the user or the system should rewrite this value-driven rule into another rule as shown in Figure 11, where '- - -' represents a special mark to the rule manage and indicates that some data should be pre-loaded. In the case, one more employee table is needed with the value of *DBMS OP* = '- - -', and the key *name* of the employee table and the attribute that contains a constant expression in the original employee table are also copied into the new added employee table. Note that the attribute that contains a constant expression in the original employee table is then removed. The interface in the RBE system pre-loads those data specified in a rule with '- - -' mark into main memory before accepts user's queries. In this example, the interface pre-loads the values of the name and the job attributes for those tuples whose job = `President` from the employee table into main memory before accepting user's queries. However, the rule shown in Figure 11 is still incorrect to represent for what we want; this rule refuses not only a query that retrieves the president's salary, but also a query that retrieves any attribute of the president. The reason is that the variable <z> shown in the salary attribute can match with any value, even the null value, i.e., *nil* in OPS5. Therefore, the correct rule for this value-driven example is shown in Figure 12, where the value of the salary attribute that the user tries to protect should be {<z> <> nil}.

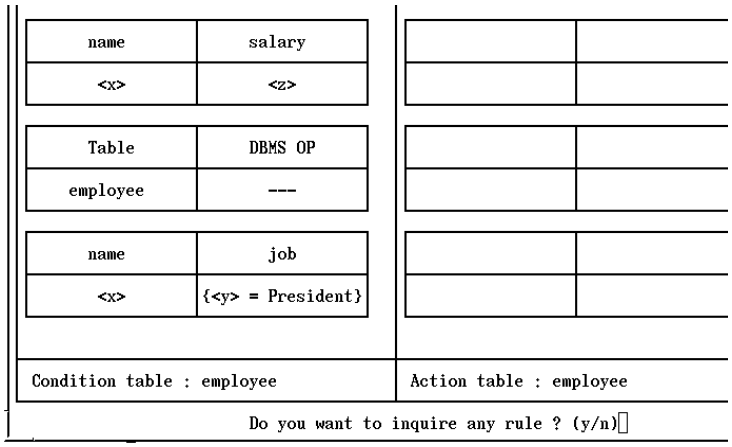


Figure 11: A rewritten value-driven rule

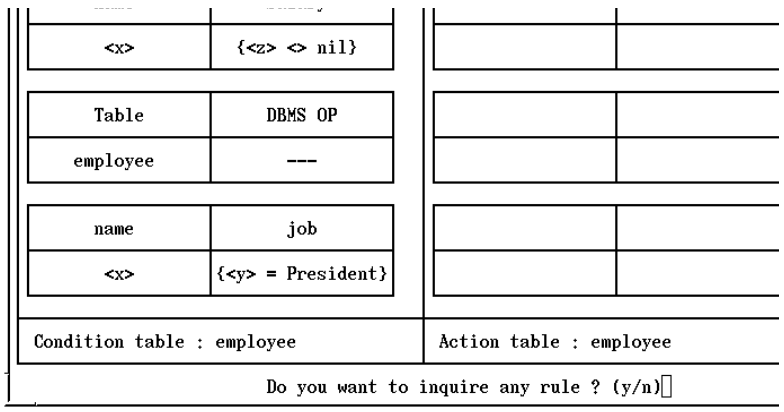


Figure 12: A correct value-driven rule

The Rule Manager

The pattern matching algorithms used in a production system well match the event-driven mechanism used in our rule manager, if we let a pattern contain an event, i.e., a query. Therefore, we use OPS5 production system to implement the rule manager. In the RBE system, all the rules have been translated into OPS5 production rules by the RMX module. Note that the syntax checking of the RBE rules also has been done in the RMX module. Moreover, the RMX module has added some rules to control the interaction between OPS5 and the interface written in C, which are discussed in details in this section.

OPS5 Production System

A production system is composed of three components: a working memory to store data, a production rule memory to store the rules and an interpreter, i.e., an inference engine, to choose applicable rules [29].

A working memory that is noted as WM is a collection of elements in OPS5. A rule contains conditional elements and action elements. Conditional elements are simple templates to be matched against data items in the WM. Action elements can use three actions to alter the contents of the working memory: (1) make: add a new element, (2) remove: delete an old element, and (3) modify: update a matched element.

An interpreter executes the *recognize-act* cycle between the execution of the applied rules. First, the interpreter executes the *pattern matching* phase to recognize those rules whose condition parts are all satisfied with the current contents of the WM and puts them into a *conflict set*. Many efficient pattern matching algorithms have been proposed [26, 27, 32, 33]. Then the interpreter executes the *conflict resolution* phase to select one of these rules from the conflict set by using a pre-defined control strategy. Finally, the interpreter executes the action part of the dominating rule to update the contents of the working memory.

Another important feature of OPS5 is that it permits external procedure/function calls inside the action elements. Depending on the version of OPS5 available, these external procedures/functions can be coded in the implementation language: Lisp, Bliss or C [28, 29]. Moreover, those external procedures/functions can also insert elements into the WM. For

```
dollar_reset(); /* clear the buffer used between OPS5 and C */
dollar_value(dollar_intern(employee)); /* insert the element name */
dollar_tab(dollar_intern(name)); /* point to an attribute */
dollar_value(dollar_intern(Jack)); /* insert a string value */
dollar_tab(dollar_intern(salary)); /* point to an attribute */
dollar_value(dollar_cvna(atoi(1000))); /* insert an integer value */
dollar_assert(); /* finished */
```

Figure 13: C codes to insert an element into the WM

example, Figure 13 shows a way to insert an element (employee ^name Jack ^salary 1000) into the WM by using C codes.

The Processing of the Rule Manager

The rule manager written in OPS5 is an output file called rbe.l of the RMX module. This rbe.l contains rules written by the users and rules to interact with the other two modules written in C: QUERY and TRANS. The QUERY module is used to input user's query, and the TRANS module is used to pre-process user's query and derived queries, which are stored in file *rule.out*, to INGRES. Figure 14 shows the flowchart of the rule manager, and Figure 15 and Figure 16 show *rbe.l* when there is only one rule called trigger in the system. The QUERY module translates user's query into an OPS5 element and then inserts the element into the WM. When an applicable rule has been triggered, the new elements produced in the action part should also be inserted into the WM for further testing.

To distinguish which user's query activates the derived queries, the rule manager assigns each user's query a number and attaches the same number to each derived query. To record the relationship between the user's query and these derived queries, a *level* attribute is used to denote the level of each query in this tree of derived queries, and a *uid* and a *pid* attributes are used to represent the unique identifier of each query and the identifier of its father. This relationship affects the execution order of those queries.

In the case that the action part of a rule is a *refuse* operation, the query which activates this rule is removed from the WM to prohibit the execution of this query, and a special mark is sent to the TRANS module to print out the query and a warning message "The query is refused". In the case that a query A activates a rule (if A then B) and a rule

Figure 14: The flowchart of the rule manager

```
(external query)
(external trans)

(literalize output_flag   ; when action = on ==> output queries
        action         ; when action = off ==> output is finished
        switch         ; when switch = off ==> seq_no should be
        seq_no)        ; increased by one

(literalize input_query   ; define the status of input-query
        start)         ; the value of start is (on, off)

(literalize employee
        dbms eno U_eno name U_name job U_job salary U_salary
        comm U_comm mgr_no U_mgr_no dno U_dno tel U_tel
        user           ; user's name
        priority       ; the priority of the rule
        level          ; the level in the tree of queries
        pid            ; the uid of its father
        uid            ; the unique identifier
        number)        ; the order among user's queries

(p initial                ; start
   (start)
-->
   (openfile ruleout |rule.out| out)
   (make input_query ^start on)
   (default ruleout write)
   (call query))            ; CALL QUERY

(p to_output              ; when no more applicable rule ==> output
   (input_query ^start on)
 -->
   (make output_flag  ^action on ^seq_no 0))

(p increase_seq_no         ; after output, increase the sequence number
   (output_flag  ^action on ^switch off ^seq_no <kk>)
   (input_query ^start on)
-->
   (modify 1 ^seq_no (compute <kk> + 1) ^switch on))
```

Figure 15: Rule Manager (rbe.l)

```
(p accept_another_query     ; the system starts to accept another query
   (input_query ^start on)
   (output_flag ^action on ^switch on)
 -->
   (modify 1)              ; change the time tag
   (remove 2)
   (closefile ruleout)
   (call trans)            ; CALL TRANS
   (openfile ruleout |rule.out| out)
   (default ruleout write)
   (call query))           ; CALL QUERY

(p exit_the_system         ; finished
   (input_query ^start off)
   (output_flag ^action on ^switch on)
-->
   (closefile ruleout)
   (call trans)            ; CALL TRANS
   (halt))

(p output_employee         ; write a query to the output file
   (output_flag ^action on ^seq_no <kk>)
   {(employee  ^dbms <dbms1> ^eno <eno1> ^U_eno <U_eno1> ^name <name1>
   ^U_name <U_name1> ^job <job1> ^U_job <U_job1> ^salary <salary1>
   ^U_salary <U_salary1> ^comm <comm1> ^U_comm <U_comm1> ^mgr_no <mgr_no1>
   ^U_mgr_no <U_mgr_no1> ^dno <dno1> ^U_dno <U_dno1> ^tel <tel1>
   ^U_tel <U_tel1> ^user <user1> ^priority <priority1> ^level <kk>
   ^uid <uid1> ^pid <pid1> ^number <number1>) <employeewrite>}
-->
    (write |employee|  |^priority| <priority1> |^level| <kk>
        |^uid| <uid1> |^pid| <pid1> |^dbms| <dbms1> |^eno| <eno1>
        |^U_eno| <U_eno1> |^name| <name> |^U_name| <U_name1>
        |^job| <job1> |^U_job| <U_job1> |^salary| <ry1>
        |^U_salary| <U_salary1> |^comm| <comm1> |^U_comm| <U_comm1>
        |^mgr_no| <mgr_no1> |^U_mgr_no| <U_mgr_no1> |^dno| <dno1>
        |^U_dno| <U_dno1> |^tel| <tel1> |^U_tel| <U_tel1> |^user| <user1>
        |^number| <number1> (crlf))
    (modify 1 ^switch off)
    (remove <employeewrite>))

(p trigger                 ; user's rule
   (input_query ^start on)
   (department ^dbms update ^dno <x> ^U_mgr_no <y>)
-->
   (bind <id>)             ; create an uid
   (make employee ^priority 3 ^dbms update ^dno <x> ^U_mgr_no <y> ^pid <idx>
        ^uid <id> ^level (compute <level> + 1) ^number <No> ))
```

Figure 16: Rule Manager (rbe.l) (continued)

```
(p rule1
   (employee ^dbms update ^name Mary ^U_salary {<x> <> nil} ^level <level>
             ^number <No> ^uid <idx>)
-->
   (bind <id>)
   (make employee ^priority 3 ^dbms update ^name John ^U_salary (compute <x> + 1000)
                  ^level (compute <level> + 1) ^number <No> ^uid <id> ^pid <idx>))

(p rule1-add
   (employee  ^dbms update ^eno nil ^name nil ^job nil ^salary nil
              ^U_salary {<x1> <> nil}
              ^comm nil ^mgr_no nil ^dno nil ^tel nil ^level <level>
              ^number <No> ^uid <idx>)
-->
   (bind <id>)
   (make employee ^priority 3 ^dbms update ^name Mary ^U_salary <x1>
                  ^level (compute <level> + 1) ^number <No> ^uid <id> ^pid <idx>))
```

Figure 17: An example of a new added rule

(if A then C), then both queries B and C, have the same value in the level attribute. In the case that a query A activates a chain of rules, for example, (if A then B), (if B then C), then query A has level = 1, query B has level = 2 and query C has level = 3. Moreover, the RMX module may add some more rules in order to make the event-driven mechanism work well. Consider the following three rules:

rule 1: if Mary's salary is updated to X, then update John's salary to X + 1000;

rule 2: if John's salary is updated to X, then update Tom's salary to X + 1000;

rule 3: if Tom's salary is updated to X, then update Joe's salary to X + 1000;

Given a query "update employee set salary = 6000", the rule manager does not activate those three rules. In order to activate those three rules, the RMX module adds one more rule for each rule whose condition part contain an *update* operation at the time when the rule is added. In this example, for rule 1, the new added rule 1' specifies that whenever employee's salary is updated to a value X and X $\neq$ *nil*, update Mary's salary to X, as shown in Figure 17. In a similar way, rule 2' and rule 3' are added by the RMX module. Now, given a user's query "update employee set salary = 6000", the sequence of rules which are activated is (3', 3, 2', 2, 3, 1', 1, 2, 3). That is, finally, Mary's salary is 6000, John's salary is 7000, Tom's salary = 8000, and Joe's salary = 9000.

After all applicable rules have been detected, the rule manager writes those queries, in addition to user's query, to an output file *rule.out*. The rule manager then calls the TRANS module to reorder those queries according to attributes level and priority, translate those queries represented in elements into SQL commands, and the TRANS module then sends those commands to INGRES. Note that we also can directly send a SQL command of a derived query from the rule manager to INGRES whenever a rule is activated. Since many rules can be activated by the same query, those derived queries may not be executed according to the derivation relationship and priority, which may result in an unexpected database state. To define clear rule execution semantics, we design the TRANS module that will be explained in details in the next section.

The Architecture

The architecture of RBE as shown in Figure 18 contains four parts: RMX, the rule manager, INGRES DBMS and the interface. The RMX module and the rule manager have been introduced in previous two sections. Therefore, in this section, we describe the functions performed in the interface that communicates with OPS5 and INGRES.

The Interface

The interface contains two modules: QUERY and TRANS. The QUERY module first processes value-driven rules, i.e., it loads related information for those value-driven rules. For example, for the rule shown in Figure 12, the QUERY module sends the following SQL command to INGRES (Select name, job from employee where job = "President"). After getting a result from INGRES, the QUERY module translate the result into the form of OPS5 elements with DBMS OP ='- - -' into the WM. After processing value-driven rules, the QUERY module starts to accept user's query either written in SQL or in a table form. The QUERY module then translates user's query into an OPS5 element, and inserts this element into the WM.

The semantics of rule execution is implemented in the TRANS module. Note that in our system, the rule testing task is performed as soon as a user inputs a query. An activated rule may trigger other rules. However, the system will not execute those activated rules

Figure 18: The architecture of the RBE system

until all applicable rules are detected. Moreover, two tasks are performed in the TRANS module: (1) determine the coupling made of the user's query and derived queries and (2) determine rule execution order. For the first task, in our RBE system, we treat each query as a transaction. Therefore, if query A triggers query B, the abortion of query B will not affect the result of query A, while the abortion of query A will result in the abortion of query B. We can also easily implement other coupling mode in the TRANS module. For the second task, the TRANS module reads the output file created from the rule manager. The output file contains user's query and derived queries represented in a form of elements. The TRANS module translates each element into a SQL command and sends each command to INGRES. Before these elements are translated into SQL commands, the TRANS module reorders these elements (i.e., queries) according to the levels of the tree and priority. The purpose of the reorder algorithm is to determine an execution order. Basically, elements are executed according to the increasing order of levels, and for the elements in the same level, they are executed according to the increasing order of priority, from low to high, since the result of a high priority rule will determine the final system state. Based on this order algorithm, it is easy to handle an exception case for a rule by assigning the general rule

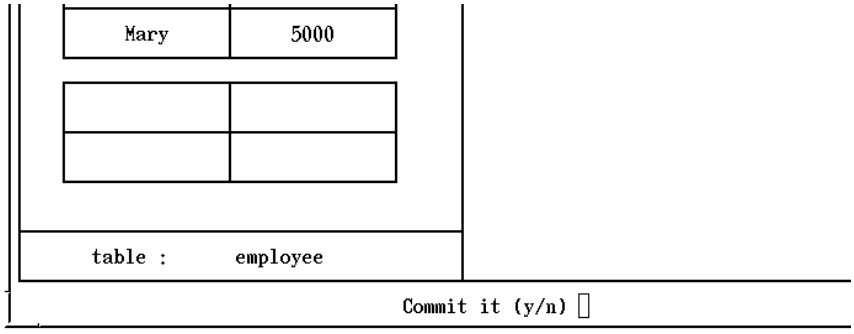
(a) (b)

Figure 19: An example of a set of applicable rules: (a) rules; (b) the element index and priority

a priority lower than the special case. After the new tree structure has been constructed, an execution order without duplicated execution is determined by tracing the tree in a depth-first search.

Consider a set of rules as shown in Figure 19-(a), where an edge marked as $tr1$ from a node Mary to a node John, for example, means that whenever Mary's salary is updated to a value X, update John's salary to X + 1000 and is recorded as rule $tr1$. Figure 19-(b) shows the element index after sorting in an increasing order of levels and priority, and priority assigned by the users, corresponding to Figure 19-(a). The execution sequence created from the reorder algorithm is (0, 1, 2, 4, 7, 9, 3, 5, 6, 8, 10, 11), where the number represents the element index. Figure 20 shows the output of the set of rules by given a query "update employee set salary = 6000 where name = 'Mary' ". In this example, since the tuple name = 'George' does not exist in the system, the related query is aborted by the system. That is, when a query is aborted, all the other queries in the subtree below this query are aborted. Obviously, the other choice is to treat the whole queries as a transaction; therefore, once one transaction is aborted, all the queries are aborted.

Each query can be executed dynamically by making use the property of run-time query




```
The derived query is (update employee set salary = 7000 where name = 'Joe')
        *** The database system process O.K. ***

The derived query is (update employee set salary = 8000 where name = 'Peter')
        *** The database system process O.K. ***

The derived query is (update employee set salary = 8000 where name = 'Amy')
        *** The database system process O.K. ***

The derived query is (update employee set salary = 9000 where name = 'George')
        ### Error from INGRES : This tuple does not exist in DB ###
       transaction id = $$OPS_129 abort

The derived query is (update employee set salary = 7000 where name = 'Tom')
        *** The database system process O.K. ***

The derived query is (update employee set salary = 8000 where name = 'Juli')
        *** The database system process O.K. ***

The derived query is (update employee set salary = 8000 where name = 'Joe')
        *** The database system process O.K. ***

The derived query is (update employee set salary = 9000 where name = 'Peter')
        *** The database system process O.K. ***

The derived query is (update employee set salary = 9000 where name = 'Amy')
        *** The database system process O.K. ***

The derived query is (update employee set salary = 10000 where name = 'George'
        ### Error from INGRES : This tuple does not exist in DB ###
       transaction id = $$OPS_135 abort
```

Figure 20: The output of a derived query

planning supported by INGRES. And the property allows the host program (C) to send a command that is unknown to the host program in advance. If the command is not a select operation, then the host program can simply send the command to INGRES. However, if the command is a select operation, then the host program must prepare enough memory storage to store returned data. Figure 21 shows the part of program which performs run time query planning in C.

The Processing of RBE

Figure 22 shows the the flowchart of the RBE system. First, the RMX module accepts user's rules, checks syntax , and translates those rules into OPS5 production rules. In addition, the RMX module adds some production rules to control the interaction between OPS5 and the interface. The output file from RMX is an OPS5 program. After the OPS5 program has been compiled, the RBE system starts to accept user's query by calling a C program QUERY. The QUERY module accepts user's query and translates the query into an element which is then inserted into the WM.

After user's query has been inserted into the WM, the control is returned back to OPS5. By the data-driven property, the rule manager activates all applicable rules. The rule manager then calls a C program TRANS to reorder all the queries and then do run time query planning. The TRANS module translates each query in the form of elements into a SQL command, and sends each command to INGRES. This module also shows information about the status of the results of user's query and derived queries on the screen. If there is no more user's query, the whole system stops.

Discussion

In this section, we discuss the criteria to distinguish event_driven rules from value_driven rules, and the properties of appropriate applications which can make use of RBE. Then we make a comparison of the RBE system with other active database systems.

```
exec sql include sqlca; /** embedded SQL */ /*  used in the TRANS module  */
exec sql include sqlda; /**/
IISQLDA *sqlda = (IISQLDA *)0; /**/
struct { int       res_length;
        char      *res_data;
} res_buf = {0,NULL};
ingres_call(cmd)
char cmd[180]; /* the command string */
{       exec sql begin declare section; /**/
             char *loname; /**/
             char err_msg[180];  /**/
        exec sql end declare section; /**/
        int row=0;
        exec sql connect rbedb; /**/
        Init_Sqlda(20); /* a procedure to allocate memory */
        loname=cmd; /* the command string */
        exec sql prepare s1 from :loname; /* compile the command string */
        exec sql describe s1 into :sqlda; /**/
        if (sqlda->sqld == 0){ /* not a selection operation */
                exec sql execute s1; /* execute the command string */
                row = sqlca.sqlerrd[2]; } /**/
        else {  exec sql declare c1 cursor for s1;    /* open a buffer */
                Print_Header(); /* a procedure to print attribute names */
                exec sql open c1 for READONLY; /**/
                while (sqlca.sqlcode == 0){ /* when execution status is O.K. */
                        exec sql fetch c1 using descriptor :sqlda; /**/
                        if (sqlca.sqlcode == 0){ /**/
                                row++;
                                Print_Row(); /* a procedure to print a row */
                                printf(\n);
                        }
                }
        }
        if (sqlca.sqlcode == 0 || ((sqlca.sqlcode == 100) && (row > 0)))
                printf(''\t*** The database system process O.K. ***\n'');
        else
                if (sqlca.sqlcode == 100)
                        fprintf(wp,''\t ### Error from INGRES : This tuple
                                does not exist in DB ###\n'');
                else { exec sql inquire_sql (:err_msg = errortext);
                       /** get the error message from INGRES **/
                       fprintf(wp,''\t ### Error from INGRES : %s ###\n'',err_msg); }
        exec sql close c1; /**/
        exec sql commit; /**/
        exec sql disconnect; /**/
        return;
}
```

Figure 21: Run-time query planning in INGRES with the host language C

Figure 22: The flowchart of RBE

Event_driven vs. Value_driven Rules

There are three cases in which a rule can be written as an event-driven rule: (1) the *where* clause is omitted, (2) the database operation is an insert operation, and (3) every expression in each condition element is of the form (^attribute variable). In these three cases, rules can be activated without accessing the data in the database. For other cases, a rule should be written in a value-driven rule style. However, if the given application has some specific properties, then rules other than the above three cases still can be activated without accessing the data in the databases. For example, this application guarantees that a user's query will always exactly match the condition part of a rule.

Properties of Appropriate Applications Based on RBE

To distinguish whether a rule should be written as an event-driven rule or a value-driven rule as described before, we now point out the properties of appropriate applications that can make use well of the event-driven mechanism supported in the RBE system. We consider this problem from two view points: the types of user's queries and the forms of rules.

First, we consider the form of user's queries. If user's queries can exactly match rules, obviously, it is always event-driven. For example, the following rule can be used for system protection.

```
(p rule1
   (employee ^dbms delete ^user <> 'chenf')
-->
   (modify 1 ^dbms refuse))
```

If the user's queries do not contain the *where* condition, i.e., the scope of the query is all the tuples in the table, and the action parts of rules do not contain "^dbms refuse", the rule can be activated by using the automatically new added rules as shown in Figure 17.

Next, we consider the forms of rules. If the *dbms* attribute of the condition parts of a rule contains *insert* only, obviously, it is an event-driven rule. For example, the following rule can be used in an inventory control.

```
(p amount-limitation
   (goods ^dbms insert ^no 5872 ^amount {<x> > 500})
-->
   (modify 1 ^dbms refuse))
```

If the *dbms* attribute can contain any operation and the rule does not contain the *where* condition, it is also an event-driven rule. For example, the following rule ensures that somebody's salary and bonus are always equal.

```
    (p rule1
  (employee ^dbms update ^U_salary <x>)
-->
  (make employee ^dbms update ^U_bonus <x>))
```

If every expression in each condition element is of the form (^attribute variable), such a rule is an event-driven rule. Appropriate applications contain version control, materialized view, log, snapshot, referential integrity and derived attribute values. Although some researchers [22] argued that a loosely-coupled system is likely to perform poorly unless the application has specific characteristics, in this section, we have shown the properties of appropriate applications which can make use of the RBE system.

A Comparison

In this Section, we make a comparison of our RBE system with several active databases, including Ariel, HiPAC, Starburst and POSTGRES, and those commercial systems that support triggers, including SYBASE, ORACLE and INFORMIX.

Ariel [18, 19] uses a subset of the POSTQUEL query language extended with a new set-oriented rule language to support production rules. Ariel rules can have conditions based on a mix of patterns, events and transitions, where a transition in Ariel is defined to be the changes of the database. HiPAC [15, 16, 17] uses the relational model for the overall framework and the nested transaction model as the framework for the execution of rules. Each rule in HiPAC is structured according to the event-condition-action paradigm. Starburst [20, 21] provides set-oriented SQL-based production rule language, which are based on the notion of transitions. POSTGRES [1, 9, 10, 11, 12, 13, 14] allows any POSTQUEL command to be tagged with three special modifiers, *always*, *refuse*, and *one-time*, which change its meaning and such tagged commands become rules. More recently, POSTGRES proposes a new rule syntax that allows users to specify event-driven rules.

In all database production rule languages, the condition part of a rule specifies a predicate or query over the data in the database [7]. Table 1 shows examples of rules in

those databases. In Ariel, the event may be omitted from a rule, in which case triggering is defined implicitly by the rule's condition. In HiPAC object-oriented active database, events can be generic database operations or type-specific operations. In Starburst, an event can be a disjunction. POSTGRES allows single explicit database triggering events. In RBE, the events in the condition part can be retrieve, insert, delete or update operations. Moreover, RBE is the only table-based language, which is more user-friendly.

The action part of a database production rule specifies the operations to be performed when the rule is triggered and its condition is satisfied. In Ariel, Starburst and POSTGRES, rule actions can be arbitrary sequences of retrieval and modification commands over any data in the database. Rule actions also may specify rollback to abort the current transaction. Rule actions in HiPAC can contain arbitrary database operations, transaction operations, rule operations, signals that user-defined events have occurred, or calls to application procedures. In RBE, the rule actions can be insert, delete, update, retrieve, refuse or show operations. Moreover, based on the loose-coupling approach, many extensions can be easily implemented in the interface. For example, rule actions can contain procedure calls or rollback operations by providing more DBMS operations in the action window and implementing those functions in the TRANS module.

Many database rule languages allow conditions in rules triggered by database modifications to refer both to the modified data and to the database state preceding the triggering event. In Ariel, the old value is referenced using the keyword *previous*. In HiPAC, the triggering event of a rule may be parameterized, and these parameters may be referred. In Starburst, a single rule triggering may involve arbitrary combinations of inserted, deleted, and updated tuples. These changes may be referenced in the condition and action part of a Starburst rule using transition tables. In POSTGRES, to reference the modified tuple before and after the triggering event, POSTGRES uses the special tuple variables *new* and *old*. In RBE, the old and the new values of an attributed A can be referenced by fields A and U_A, respectively.

In Ariel and POSTGRES, rules have numeric priorities. In Starburst, rules are partially ordered. In HiPAC, multiple triggered rules are executed concurrently using an extended nested transaction model. HiPAC rules may have relative ordering, and this

Table 1: A Comparison

ordering is used to influence the serialization order of concurrently executing nested sub-transactions. In RBE, rules also have numeric priorities.

There are four basic techniques which can be applied in a rule manager [14]: (1) brute force, (2) discrimination networks, (3) marking, and (4) query rewrite. Brute force entails maintaining a list of all rules that affect each table in a database. Then, each individual update is matched against the condition part of each rule in the list to determine which must be activated. Discrimination networks, such as RETE and TREAT, have been widely used in expert system shells to speed up this search. The RETE algorithm compiles the conditions of productions into a binary discrimination network. The third technique is to utilize a marking system, in which each rule is processed against the database and every record satisfying the event qualification is identified. Each record is marked with a flag identifying the rule to be activated. A fourth implementation technique is called query rewrite. In this case, each applicable rule is substituted into the user command to produce a modified command.

For testing rule conditions, Ariel makes use of a discrimination network composed of a special data structure for testing single-relation selection conditions and a modified version of the TREAT algorithm, called A-TREAT, for testing join conditions [18]. Ariel is unique in its use of a selection-predicate index that can efficiently test predicates of rules on any attribute of a relation.

The HiPAC system applies the following techniques in the rule manager [16]: (1) multiple condition optimization, (2) materialization and maintenance of intermediate results, (3) identification of readily ignorable events, (4) incremental evaluation, and (5) using knowledge of the action parts of rules. Central to the concept of condition monitoring is the graph abstraction. The graph abstraction is comparable to the RETE network structure used in OPS5. However, the graph abstraction differs from the RETE network in several aspects. For example, instead of the recognize-act cycle in the RETE network, HiPAC intends to concurrently evaluate several conditions using several sub-graphs.

In the implementation of the Starburst rule manager, rule conditions and actions may refer to transition tables, including inserted, deleted, new-updated, and old-updated tables, which are logical tables reflecting the changes to the rule table that have occurred during

the triggering transition [20].

POSTGRES uses several techniques for rule optimization [10, 11, 12], such as indexing, cashing, lazy-evaluation, and eager-evaluation. Besides these, it supports tuple level processing, i.e., the marking strategy, and the query rewrite strategy. A marking strategy is implemented based on individual record accesses and updates to the database. It will work well if there are many rules, each affecting only a few instance. However, it is impossible for the query optimizer to construct an efficient execution plan for a chain of rules that are activated. Moreover, the implementation of the marking mechanism is very complex, and it may be incorrect [11]. To support a rule like "Joe's salary should be the same as Fred's", four kinds of markers on different fields or indexes are needed. Therefore, it has been a big challenge to ensure that markers are correctly installed and appropriate actions are taken when record accesses and updates occur. Moreover, for rules which cover many instances but not a significant fraction of all instances, the marking implementation is not very space efficient. For the query rewrite implementation, a rule could be applied by converting a user's query to an alternate form prior to execution. This transformation is performed between the query language parser and the optimizer. Support for views is done in this way along with many of the proposals for a recursive query support. It will work well when there are a small number of rules on any given constructed type and most rules cover the whole constructed type. However, the number of queries as well as the complexity of their qualifications increases linearly with the number of rules, which will result in bad performance unless multiple query optimization techniques are applied. When applicable, the compile-time approach of query rewrite can be considerable more efficient than the run-time approach of tuple-level marking processing. When the markers are escalated to the constructed type, the query rewrite strategy is replaced. Consequently, these two implementation strategies have their own advantages/disadvantages, and are complementary with each other.

Compared to POSTGRES, in RBE, an event-driven rule processing can cover the cases that a query rewrite strategy can perform well, and a value-driven rule processing can cover the cases that a marking strategy can perform well. However, the implementation of the query rewrite strategy, needs four steps [12]. And, the query rewrite strategy may send

some extra queries to DBMS and requires multiple query optimization strategy to improve the performance [10], while RBE will activate only those applicable rules by making use of well developed pattern matching algorithms no matter how complex a rule is, no matter the number of rules is large or small, and no matter the overlapped scope of rules is large or small.

Compared with these active database systems, including Ariel, HiPAC, Starburst and POSTGRES, the RBE rule system is not designed from scratch. The RBE system loosely couples the OPS5 and INGRES database system. For the other systems, they are designed from scratch; i.e., they tightly couple the rule manager and the database. Moreover, RBE applies the *event-driven* mechanism to design the system, and makes use of well-developed pattern-matching algorithms used in a production system as the rule manages, as compared to the marking or transition-table-based mechanisms applied in some other active database systems.

The semantics of a database production rule language determines how rule processing will take place at run-time once a set of rules has been defined, including how rules will interact with the arbitrary database operations and transactions that are submitted by users and application programs. In Ariel, rule processing is invoked automatically at the end of each transition and the rules actually consider the net effect of the modifications in the transition rather than the individual modifications. Rule processing in HiPAC is invoked whenever any event occurs that triggers one or more rules. In Starburst, rule processing is invoked automatically at the end of each user transaction that triggers one or more rules. In addition, users can invoke rule processing within transactions by issuing special commands. Like Ariel's, Starburst's rules consider the net effect of sets of modification, rather than the individual modification. In POSTGRES, rule processing is invoked immediately after any modification to any tuple that triggers and satisfies the condition of one or more rules. This case sometimes is referred to as tuple-oriented rule processing, as opposed to Ariel's set-oriented rule processing. In RBE, rule processing is invoked immediately after user's query is input and before the query is executed.

Coupling modes determine how rule events, conditions, and actions relate to database transactions. There are three coupling modes: *immediate* indicating immediate execution:

deferred indicating execution at the end of the current transaction, and *decoupled* in dictating execution in a separate transaction. Ariel and Starburst support *deferred* mode, while POSTGRES support *immediate* coupling mode only. HiPAC proposed a general execution model, in which the rule definer has the flexibility of deciding whether or not the conditions and actions should execute in the triggering transaction. In RBE, we can implement these different options in the interface, i.e., the TRANS module although we have implemented the immediate mode so far. All the systems allow *cascaded execution* of rules.

Commercial DBMSs have been introducing support for triggers at various levels. However, there are usually some limitation. The trigger events can only be built-in SQL operations, like update, insert, delete, on a single base table. Triggers over views are not allowed. Trigger can only be part of the triggering transactions and triggers cannot be nested. For example, in SYBASE, the condition part of a trigger can only refer to one table, and only the base table; a trigger cannot be created on a view. Only one trigger can be associated with an operation on a table. The action part of a trigger is limited to a sequence of SQL statements. Furthermore, triggering is limited to one level, where the triggered actions themselves do not cause triggers to be activated. Similarly, in INFORMIX, a user can create a trigger on a table in the current database; a user cannot create a trigger on a temporary table, a view, or a system catalog table. Moreover, in INFORMIX, if a user defines more than one update trigger event on a table, the column lists of the triggers must be mutually exclusive. For example, in Figure 23, trig2 is illegal because its column list includes stock_num that is a triggering column in trig1. In ORACLE, users must first decide what kinds of events, such as exiting a certain field, which the triggers should be activated, and write triggers with explicit specification of those events. There are totally 17 trigger events, for example, pre-field, post-change and post-field are three field triggers. There are also some restrictions in using SQL commands in a trigger. For example, a user can only use SELECT commands in a post-change trigger. In RBE, a rule can refer any number of tables, or views in either the condition part or the action part. Moreover, rule execution can be nested.

```
CREATE TRIGGER trig1 UPDATE OF item_num, stock_num ON items
FOR EACH ROW(EXECUTE PROCEDURE proc1);
CREATE TRIGGER trig2 UPDATE OF order_num, stock_num ON items
FOR EACH ROW(EXECUTE PROCEDURE proc2);
```

Figure 23: An illegal trigger in INFORMIX

Conclusions

In this paper, we have presented the design of a loosely-coupled active database system called RBE. In this system, a user-friendly Rule-By-Example language, which has similar syntax as QBE, is provided. To reduce the intensive interaction between the DBMS and the rule manager, the system has applied an event-driven mechanism in which rules are activated before the user's query accesses the data in the database. An event-driven mechanism well matches the pattern-matching algorithms used in a production system, if we let a pattern contain an event, i.e., a query. Based on this event-driven mechanism, we have presented an architecture which loosely couples a database system and a production system to construct the RBE system. We have showed that a loose-coupling system, like our RBE, can provide a higher flexibility than a tight-coupling system. Moreover, as the efficiency of the pattern matching algorithms will be further improved by the researchers working in the research area of artificial intelligence, the performance of our RBE system will also be improved. The architecture used in this system also has shown the applicability of constructing an active database system by integrating any production system and any database system. Moreover, the proposed technique could be used as an implementation method for a query-rewrite rule system inside a DBMS server, not using a layered approach. That is, the potential use of rule-based query rewrite processing in a tightly-coupled arrangement could improve the performance of the task of query rewrite trigger processing in those systems which have ad-hoc query rewrite systems.

In [10], they state that an interface between rule processing systems and database systems can perform well if the rule system can easily identify a small subset of the data to be loaded into the working memory of the rule manager. In RBE, when a rule is event-driven, no data should be loaded into the working memory, but when a rule is value-driven, we can identify the needed subset of the data to be loaded into the working memory.

For some functions in like views maintenance, snapshots, derived attribute values and integrity constraints, our loosely-coupled RBE system can work well and straightforward. Our approach has shown that a loose-coupling approach is economical and flexible.

Future research can be done in several directions. One of the directions is to extend the RBE system to support a distributed environment. In a distributed active database system, two parts can be distributed: data and rules [34, 35]. The most difficult part is how to activate rules. Moreover, as the rule base for an application grows, problems resulting from unexpected interactions among rules become more likely to occur. Therefore, the tools and techniques used to develop and manage large, complex rule bases are also needed [36, 37]. Furthermore, the workload of an active DBMS consists of two types of actives: externally generated tasks submitted by users and rule management tasks caused by the system. How to define a transaction boundary that can provide a good system performance given varying levels of data contention, rule complexity and data sharing between externally submitted tasks and the resulting rule management tasks, is also an important research direction [38, 39].

Acknowledgements

This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-82-0408-E-110-005.

References

[1] M. Stonebraker, "The Implementation of Integrity Constraints and Views By Query Modification," *Proc. of ACM-SIGMOD Conf. on Management of Data,* pp. 65-78, 1975.

[2] O. P. Buneman and E. K. Clemons, "Efficiently Monitoring Relational Databases," *ACM Trans. on Database Systems,* Vol. 4, No. 3, pp. 368-382, 1979.

[3] K. R. Dittrich, A. M. Kotz and J. A. Mulle, "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Database," *SIGMOD Record,* Vol. 16, No. 3, pp. 22-36, 1986.

[4] A. Cornelio and S. B. Navathe, "Using Active Database Techniques for Real Time Engineering Applications," *Proc. of 1993 IEEE International Conf. on Data Engineering,* pp. 100-107, 1993.

[5] C. F. Eick and P. Werstein, "Rule-Based Consistency Enforcement for Knowledge-Based Systems," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 5, No. 1, pp. 52-64, Feb. 1993.

[6] U. Jaeger and J. C. Freytag, "An Annotated Bibliography on Active Databases," *SIGMOD Record,* Vol. 24, No. 1, pp. 58-69, March 1995.

[7] W. Kim, *Modern Database Systems,* Chap. 21, pp. 435-455, Addison Wesley, 1995.

[8] S. Chakravarthy, "Early Active Database Efforts: A Capsule Summary," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 7, No. 6, pp. 1008-1010, Dec. 1995.

[9] M. Stonebraker and L. Rowe, "The Design of POSTGRES," *Proc. of ACM-SIGMOD Conf. on Management of Data,* pp. 289-300, 1986.

[10] M. Stonebraker, E. Hanson and S. Potamianos, "The POSTGRES Rule Manager," *IEEE Trans. on Software Engineering,* Vol. 14, No. 7, pp. 897-907, 1988.

[11] M. Stonebraker, L. A. Rowe and M. Hirohama, "The Implementation of POSTGRES," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 2, No. 1, pp. 125-142, 1990.

[12] M. Stonebraker, A. Jhingran, J. Goh and Spyros Potamianos, "On Rules, Procedures, Caching and Views in Data Base Systems," *Proc. of ACM-SIGMOD Conf. on Management of Data,* pp. 281-290, 1975.

[13] M. Stonebraker and G. Kemnitz, "The POSTGRES Next-Generation Database Management System", *Comm. of the ACM,* Vol. 34, No. 10, pp. 78-92, 1991.

[14] M. Stonebraker, "The Integration of Rule Systems and Database Systems," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 4, No. 5, pp. 415-423, 1992.

[15] U. Dayal, "Active Database Management System," *Proc. of the International Conf. on Data and Knowledge Bases,* pp. 150-169, 1988.

[16] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M.J. Cary, M. Livny and R. Jauhari, "The HiPAC Project: Combining Active Database and Timing Constraints," *SIGMOD Record,* Vol. 17, No. 1, pp. 51-70, 1988.

[17] D. McCarthy and U. Dayal, "The Architecture of an Active Database Management System," *Proc. of ACM SIGMOD Conf. on Management of Data,* pp. 215-224, 1989.

[18] E. N. Hanson, "Rule Condition Testing and Action Execution in Ariel," *Proc. of ACM SIGMOD Conf. on Management of Data,* pp. 49-58, 1991.

[19] E. N. Hanson, "The Design and Implementation of the Ariel Active Database Rule System," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 8, No. 1, pp.157-172, Feb. 1996.

[20] l. Haas, G. L. W. Chang, J. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey and E. Shekita, "Starburst Midflight: as the Dust Clears," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 2, No. 2, pp. 143-160, 1990.

[21] G. M. Lohman, B. Lindsay, H. Pirahesh and K. B. Schiefer, "Extensions to Starburst: Objects, Types, Functions and Rules," *Comm. of the ACM,* Vol. 34, No. 10, pp. 94-109, Oct. 1991.

[22] M. Stonebraker, ED., *Readings in Database Systems,* Los Altos, CA: Morgan-Kaufman, pp. 503-506, 1988.

[23] A. P. Sheth, "Does Loose AI-DBMS Coupling Stand a Chance," *Proc. of 1989 IEEE International Conference on data Engineering*, pp. 252-254, 1988.

[24] L. Kerschberg, "The Role of Loose Coupling in Expert Database System Architectures," *Proc. of 1989 IEEE International Conference on data Engineering*, pp. 255-256, 1988.

[25] D. S. Parker. "Integrating AI and DBMS Through Stream Processing," *Proc. of 1989 IEEE International Conference on Data Engineering*, pp. 259-260, 1988.

[26] C. Forgy, "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence,* Vol. 19, No. 1, pp. 17-37, 1982.

[27] D. Miranker, "TREAT: A Better Match Algorithm for AI Production Systems," *Proc. of AAAI Conference on Artificial Intelligence,* pp. 33-42, 1987.

[28] D. Kalp, M. Tambe, A. Gupta, C. Forgy, A. Newell, A. Acharya, B. Milnes and K. Swedlow, "Parallel OPS5 User's Manual," *Technical Report CMU-CS-88-187,* Dept. of Computer Science, Carnegie Mellon Univ. 1988.

[29] L. Brownston, R. Farrel, E. Kart and N. Martin, *Programming Expert Systems in OPS5,* Addison-Wesley Publishing Company, 1985.

[30] M. M. Zloof, "Query-By-Example: A Database Language," *IBM System Journal,* Vol. 16, No. 4, pp. 324-343, 1977.

[31] C. J. Date, *An Introduction to Database Systems,* Vol. 1, Addision-Wesley Publishing Company, Reading Massachusetts, 6th edition, 1995.

[32] T. Ishida, "An Optimization Algorithm for Production Systems," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 6, No. 4, pp. 549-558, August 1994.

[33] A. J. Pasik, "A Source-to-Source Transformation for Increasing Rule-Based System Parallelsim," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 4, No. 4, pp. 336-343, August 1992.

[34] A. Gupta and J. Widom, "Local Verification of Global Integrity Constraints in Distributed Databases," *Proc. of 1993 ACM SIGMOD,* pp. 49-58, 1993.

[35] I. M. Hsu, M. Singhal and M. T. Liu, "Distributed Rule Processing in Active Databases," *Proc. of 1992 IEEE International Conf. on Data Engineering,* pp. 106-113, 1992.

[36] D. A. Brant and D. P. Miranker, "Index Support for Ruled Activation," *Proc. of 1993 ACM SIGMOD,* pp. 42-48, 1993.

[37] T. Sellis, C. C. Lin and L. Raschid, "Coupling Production System and Database Systems: A Homogeneous Approach," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 5, No. 2, pp. 240-256, April 1993.

[38] M. J. Carey, R. Jauhari and M. Livny, "On Transaction Boundaries in Active Database: A Performance Perspective," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 3, No. 3, pp. 320-336, September 1991.

[39] M. Hsu, R. Ladin and D. R. McCarthy, "An Execution Model for Active Data Management Systems," *Proc. of International Conf. on Data and Knowledge Base,* pp. 171-179, 1988.