ON RULES, PROCEDURES, CACHING and VIEWS IN DATA BASE SYSTEMS

Michael Stonebraker, Anant Jhingran, Jeffrey Goh and Spyros Potamianos
EECS Dept.
University of California, Berkeley

Abstract

This paper demonstrates that a simple rule system can be constructed that supports a more powerful view system than available in current commercial systems. Not only can views be specified by using rules but also special semantics for resolving ambiguous view updates are simply additional rules. Moreover, procedural data types as proposed in POSTGRES are also efficiently simulated by the same rules system. Lastly, caching of the action part of certain rules is a possible performance enhancement and can be applied to materialize views as well as to cache procedural data items. Hence, we conclude that a rule system is a fundamental concept in a next generation DBMS, and it subsumes both views and procedures as special cases.

1. INTRODUCTION

Most commercial relational systems support the concept of relational views [STON75]. Hence, a virtual relation can be defined to the data manager, e.g:

```
define view TOY_EMP as
retrieve (EMP.name, EMP.age, EMP.salary)
where EMP.dept = ‘‘toy’’
```

Then, all queries and many updates to TOY_EMP can be mapped to commands on the underlying base relation, EMP. Unfortunately, there are abundant examples of views for which certain updates are impossible to successfully map [CODD74]. Commercial systems simply issue error messages for such commands, and this shortcoming limits the utility of views. Clearly, the ability to specify update semantics for ambiguous updates to views would be a desirable enhancement.

Recently, several authors have proposed materializing views for augmented performance. In this case, the DBMS would keep a physical instantiation of a view such as TOY_EMP. Hence, queries to TOY_EMP can usually be processed much faster than without a materialization. However, updates to TOY_EMP will require more expensive processing because each update must be both mapped to underlying base relations and processed on the materialization. See [BLAK86, ROUS87] for details on this approach.

There are several extensions to the view mechanism that might be desirable. First, it would be nice if a view could be specified by multiple query language commands, e.g:

```
define view SHOE_EMP as
```

This research was sponsored by the Army Research Organization Grant DAAL03-87-0083 and by the Defense Advanced Research Projects Agency through NASA Grant NAG2-530.

```
retrieve (EMP.name, EMP.salary, EMP.age) where EMP.dept = ‘‘shoe’’
retrieve (NEWEMP.name, NEWEMP.salary, NEWEMP.age) where NEWEMP.dept = ‘‘shoe’’
```

Here, SHOE_EMP is defined to be the union of two commands. Although it is possible to express this view in SQL using the union operator, updates to union views are disallowed in commercial SQL implementations. Also, it is easy to propose collections of commands whose target lists are not union compatible, and therefore that are not expressible as SQL views. Lastly, it would also be nice if **partial** views were supported, i.e. a view that is defined by a collection of stored tuples as well as by one or more view definitions to materialize the remainder of the tuples.

Some researchers have proposed that procedural data types be supported in a data base system [STON87]. In this case, a column of a relation would contain data items, each of which is an unrestricted collection of query language commands. An example of this capability is storing information about hobbies of employees, e.g:

```
EMP (name, hobbies)
```

In the hobbies field we record all information about each hobby that an employee engages in. One way to model this situation is to construct one relation in the data base for each possible hobby indicating the names of the employees who practice the hobby and relevant data about their passtime. Example relations might include:

```
SOFTBALL (name, position, average)
JOGGING (name, miles, best-time)
```

Then each value in the hobbies field is a procedure consisting of a collection of query language commands retrieving relevant hobby data. Consequently, the appropriate procedure for an employee Joe who practices both softball and racing might be called foobar and would be:

```
retrieve (SOFTBALL.all) where SOFTBALL.name = "Joe"
retrieve (JOGGING.all) where JOGGING.name = "Joe"
```

Hence, the appropriate insert to EMP would be:

```
append to EMP (name = "Joe", hobbies = foobar)
```

A special case of a procedural field is the situation where each tuple contains a procedure of the form:

```
retrieve (relation.all)
```

In this case, the value of the field is the indicated relation and the procedure efficiently simulates a nested relational data structure [DADA86].

In [STON86], it was also suggested that a DBMS optimize procedural fields by precomputing the value of the procedure, rather than waiting for the user to request evaluation.

Lastly, [ROWE87] proposed a special case of procedural fields, namely that a column of a relation contain the same procedure in each tuple, differing only in the value of one or more parameters. For example, consider the following DEPT relation:

```
DEPT (dname, floor, composition)
```

Here, dname and floor are conventional fields while composition is intended to be the collection of EMP records for employees in the given department. In this case, composition can be declared to be the following procedure:

```
retrieve (EMP.all) where EMP.dept = $.dname
```

Hence, each row of DEPT has the same procedure, namely the above query, differing only in the value of the parameter, $.dname, which is available in the dname field in each tuple. As can be noted, parameterized (or special) procedures are an efficient means to support the type "collection of tuples in another relation".

In this paper we indicate that all the following concepts:

views

special semantics for updating views
materialized views
partial views
procedures
special procedures
caching of procedures

can be subsumed by one general purpose rules system. Hence, we recommend that implementors concentrate on a single powerful rules system and then simulate all of these concepts using the rules system. This is exactly what we are doing in Version 2 of POSTGRES.

Consequently, in Section 2 we present the rules system that we are building for Version 2. Section 3 continues with the two alternate implementations that we are constructing for activating rules. Then, in Section 4 we consider caching the action portion of certain rules. Section 5 indicates how view processing can be effectively layered on this rules system. Lastly, Section 6 concludes with the implementation of procedures and special procedures as particular rules.

2. THE NEW POSTGRES RULES SYSTEM

Although the first version of POSTGRES proposed using a paradigm for rules in which a command was logically **always** in execution or **never** in execution, the second POSTGRES rules system (PRS2) takes a more traditional production system approach to rules [ESWA76, STON82, DELC88, HANS89, MCCA89, WIDO89]. PRS2 has points in common with these other proposals; in fact the syntax is quite close to that of [HANS89, WIDO89]. However, the main contribution of this paper is to show how views and procedures can be subsumed under a rule paradigm. This section briefly discusses the syntax of rules in PRS2.

A rule has the form:

DEFINE RULE rule-name [AS EXCEPTION TO rule-name]
ON event TO object [[FROM clause] WHERE clause]
THEN DO [instead] action

Here, event is one of:

retrieve
replace
delete
append
new (i. e. replace or append)
old (i.e. delete or replace)

Moreover, object is either:

a relation name
or
relation.column, ..., relation.column

The FROM clause and WHERE clause are normal POSTQUEL clauses with no additions or changes. Lastly, the action portion of the DO clause is a collection of POSTQUEL commands with the following change: NEW or CURRENT can appear instead of a tuple variable whenever a tuple variable is permissible in POSTQUEL.

The semantics of a PRS2 rule is that at the time an individual tuple is accessed, updated, inserted or deleted, there is a CURRENT tuple (for retrieves, replaces and deletes) and a NEW tuple (for replaces and appends). If the event and the condition specified in the ON clause are true for the CURRENT tuple, then the action part of the rule is executed. First, however, values from fields in the CURRENT tuple and/or the NEW tuple are substituted for:

CURRENT.column-name

```
NEW.column-name
```

The action part of the rule executes with same command and transaction identifier as the user command that caused activation.

For example, consider the following rule:

```
define rule example_1
on replace to EMP.salary where EMP.name = ‘‘Joe’’
then replace EMP (salary = NEW.salary) where EMP.name = ‘‘Sam’’
```

At the time Joe receives a salary adjustment, the event will become true and Joe’s current tuple and proposed new tuple are available to the execution routines. Hence, his new salary is substituted into the action part of the rule which is subsequently executed. This propagates Joe’s salary on to Sam.

There is no requirement that the event and action be the same kind of command. Hence, consider a second rule:

```
define rule example_2
on retrieve to EMP.salary where EMP.name = ‘‘Joe’’
then replace EMP (salary = CURRENT.salary) where EMP.name = ‘‘Bill’’
```

This rule ensures that Bill has a salary equal to Joe’s whenever Joe’s salary is accessed.

Each rule can have the optional tag "instead". Without this tag the action will be performed in addition to the user command when the event in the condition part of the rule occurs. Alternately, the action part will be done instead of the user command.

For example, if Joe is not allowed to see the salary of employees in the shoe department, then the following rule is the appropriate specification:

```
define rule example_3
on retrieve to EMP.salary where EMP.dept = ‘‘shoe’’ and user() = ‘‘Joe’’
then do instead retrieve (salary = null)
```

At the time the event is true, there will be a current tuple for some member of the shoe department. The action part of the rule specifies that this person’s salary is not to be returned; instead a null value should be inserted before the tuple is returned to higher level software for further processing. Notice, that insertion of a real value in the action part of the rule would allows POSTGRES to **lie** to Joe about the salaries of members of the shoe department by returning an incorrect value.

It is also possible to update the CURRENT tuple or the NEW tuple using a rule. Hence, a somewhat more obscure way to express the above rule is:

```
define rule example_4
on retrieve to EMP.salary where EMP.dept = ‘‘shoe’’ and user() = ‘‘Joe’’
then do replace CURRENT (salary = null)
```

This rule has the same condition as the previous one; however it specifies that the current tuple is to be modified by changing the salary field to null. This will return a null value to the requesting user.

Each rule is given a rule-name, which is used to remove the rule by name when it is no longer needed. Lastly, each rule can be optionally designated to be an **exception** to another rule. Consider, for example, the following rule:

```
define rule example_5 as exception to example_4
on retrieve to EMP.salary where EMP.name = ‘‘Sam’’ and user() = ‘‘Joe’’
then do replace CURRENT (salary = 1000)
```

Suppose Sam is a member of the shoe department. In this case, example_4 would return null for his salary while example_5 would return 1000. Clearly, returning both null and 1000 is inappropriate, so the exception clause in example_5 indicates that 1000 should be returned.

Exceptions are supported by converting the parent rule, in this case example_4, to have an event qualification of

```
event_qualification and not event_qualification_of_offspring
```

Then the parent rule is removed and reinstalled, while the offspring rule is inserted in the normal manner. It is an error for the offspring rule to have a different event-type and event-object from its parent. If an exception hierarchy is present, then the above procedure must be repeated up the hierarchy until no further exceptions are encountered.

We now indicate three final examples of PRS2. Consider the rule that Sam should be given any raise that Joe receives as noted in example_1. Suppose further that we want a rule to ensure that example_1 is the **only** way that Sam's salary should be adjusted. This is specified in example_6.

```
define rule example_6
on replace to EMP.salary where EMP.name = ‘‘Sam’’ and query() != example_1
then do instead
```

This rule will disallow any command except the rule example_1 from changing the salary of Sam. Example_1 and example_6 together ensure that Sam has the same salary as Joe.

Consider the desire to construct a security audit whenever any salary is accessed. This is easily expressed by the following rule:

```
define rule example_7
on retrieve to EMP.salary
then do append to AUDIT (accessor = user(), object = CURRENT.name, value = CURRENT.salary)
```

Lastly, consider the rule in example_8:

```
define rule example_8
on retrieve to TOY_EMP
then do instead retrieve (EMP-OID = EMP.OID, EMP.name, EMP.age, EMP.salary)
where EMP.dept = ‘‘toy’’
```

This specifies that when a user performs a retrieve to TOY_EMP that instead he should be given the result of the query in the action part of the rule, namely the employees in the toy department. Clearly, example_8 corresponds to a view definition.

3. IMPLEMENTATION

There are two implementations for rules in PRS2. The first is through **tuple level** processing deep in the executor. This rules system is called when individual tuples are accessed, deleted, inserted or modified. The second implementation is through a **query rewrite** implementation. This module exists between the parser and the query optimizer and converts a user command to an alternate form prior to optimization. The tuple level implementation can process all PRS2 rules while the query rewrite implementation is used to more efficiently process a subset of PRS2. In the rest of this section we discuss each implementation.

3.1. Tuple Level Rule System

This rules system can be invoked for any PRS2 rule, and activation of rules occurs in one of two ways. First, the execution engine may be positioned on a specific tuple and a rule awakened. In this case NEW and CURRENT take on values from the new and current tuple, and the rule manager executes the appropriate action. This may include running other POSTQUEL commands and modifying values in NEW or CURRENT. The second way the rule manager is called is when a relation is opened by the executor. This will occur in example_8 above when a scan of TOY_EMP is initiated and the rule manager must be called to materialize specific tuples from another relation. This corresponds to materializing the tuples in a view one-by-one for any user query on the view. A more efficient scheme will be discussed in the next subsection.

Moreover, when the executor calls the rule manager to activate a retrieval rule, it knows how many tuples it wants returned. For example, consider the following rule:

```
define rule example_9
```

```
on retrieve to EMP.salary where EMP.name = "Joe"
then do instead retrieve (EMP.salary) where EMP.name = "John"
```

Even if there are multiple John's in the data base, the executor only wants one salary returned. On the other hand, the executor sometimes wants multiple tuples returned as in example_8 above. Hence, the executor will alert the rule manager which situation exists and coordinate the protocol for a multiple tuple return.

Execution of an activated rule can occur either before any further tuples are processed from the user's query plan, or alternatively, information about the activated rule is put in a *rule agenda*, and rule execution takes place at the conclusion of the user's query. In the latter case, the rule is run with the same command identifier as the user's query, and therefore cannot see any changes made by the the user command, apart from the new values of the tuple that triggered the rule. Alternately, it would be possible to run the rule with a separate command identifier and thereby allow each rule to see all changes of the user command and preceding rules. It would also be reasonable to defer execution of rules until the end of the transaction. We are not actively exploring either of these latter options.

When the executor processes a define rule command which will be implemented by the tuple level implementation, information about the rule must be inserted in the system catalogs and *rule locks* must be placed at either the relation level or the tuple level. There are three types of rule locks, namely *Event*, *Import* and *Export* locks.

Event locks are placed on appropriate fields in at least all tuples of the relation that appears in the event specification of a rule and satisfy the event qualification. For example consider the rule:

```
define rule example_10
on retrieve to EMP.salary
where EMP.age > 50 and EMP.dept = DEPT.dname and DEPT.floor = 1
then do instead retrieve (salary =10000)
```

Here, *Event* locks should be placed on the salary field of all employees who are more than 50 years old and work in a first floor department. *Event* locks are tagged with the type of event which will activate the action, in this case a retrieve along with the identifier of the rule. When an appropriate event occurs on a field marked by an appropriate *Event* lock, the qualification in the rule is checked and if true, the action is executed.

Apart from putting *Event* locks in the right tuples at rule definition time, we must make sure that these locks remain on the right tuples even when updates change the database state. For instance, in the case of the rule in example_10, if a new employee is added, we must check whether it satisfies the rule qualification, and if yes put the appropriate *Event* locks in his tuple. Alternately, if a department is transferred from the second floor to the first one, then we have to add locks to all the employees working there.

To achieve this effect, we use *stub* records. These can be either *index stub* records or *relation stub* records. The first ones are inserted at at each end of the scan of any index, and each intervening index record between these *stub* records must also be marked with the appropriate lock. If all index records on a page are locked, then the corresponding rule lock can be escalated to the parent page to save space. Details of this scheme appear in [KOLO89]. Whenever a new tuple is added to the relation, or the value of the indexed field is updated, a record will be added to the index. By looking at the locks in the index records adjacent to the one just inserted, the locks that should be put on the new tuple and index record can be deduced. This mechanism is more fully discussed in [STON88]. The *relation stub* records are put at the relation level, and every time a new tuple is inserted into a relation, we check all its *stub* records to deduce the locks that must be added to the new tuple.

The locks are usually put at each individual tuple. However, in order to save space, if more than a cutoff number, CUTOFF-1, of locks are set, then the lock can be escalated to a whole column of a relation and placed in the appropriate record of the system catalogs.

Event locks identify the rule that must be awakened when a particular event occurs. However, to keep the *Event* locks correct as updates occur, we require two additional kinds of locks. For example, in example_10 corrective action must be taken if the name of a department or its floor changes. To support such actions, we require *Import* and *Export* locks. In general, lets assume that the rule qualification has the following form:

```
on event to $R_n$. y
where
   $R_n$.x <op> $R_{n-1}$.y and
   $R_{n-1}$.x <op> $R_{n-2}$.y and
   .....
   $R_2$.x <op> $R_1$.y and
   $R_1$.x <op> C
```

where "<op>" is any binary operator, and "C" a constant. If the rule qualification is more complex, we can always ignore some of its clauses to construct a qualification of the above form. Of course that means that we might install more locks than necessary, because more tuples will satisfy the new qualification than the original one, but we believe that this is a reasonable tradeoff between efficiency and simplicity of implementation.

On every relation R_i (i = 1,2, ...n) we must place an *Import* lock on the "x" field and an *Export* lock on the "y" field of all tuples that satisfy the qualification:

Q_i : R_i.x <op> R_{i-1}.y and ... and R_1.x = C

and ensure that these locks will be placed on all modified or inserted tuples that satisfy Q_i This can be done in the same ways as for the *Event* locks by using *stub* records.

Specifically, when a tuple is inserted in R_i, or we modify the "x" field of a tuple so that qualification Q_i is now satisfied, we must add an *Import* lock to the "x" field and an *Export* lock to the "y" field of this tuple. Moreover, if *Y* is the value of the "y" field of the tuple, then it is necessary to add *Import* and *Export* locks to all the tuples of relation R_{i+1} which satisfy the qualification : R_{i+1}.x = *Y*. This process will be continued iteratively on R_{i+2} ...

When a tuple is deleted from R_i which had a lock on it, or the "x" field is modified so that the value no longer satisfies Q_i, then it is necessary to delete the *Import* and *Export* locks from the tuple. Moreover, if there exists no other tuple in R_i which satisfies the qualification : R_i.y = *Y* and Q_iq. we must propagate the lock deletion to all the tuples of R_{i+1} where R_{i+1}.x = *Y*, and iteratively on R_{i+2}, ...

Finally if we modify the "y" field of a locked tuple of R_i, we have to delete locks using the old value of "y" and then add locks using the new value of "y".

3.2. Query Rewrite Implementation

The tuple level implementation will be extremely efficient when there are a large number of rules, each of whose events covers a small number of tuples. On the other hand, consider example_8

```
define rule example_8
on retrieve to TOY_EMP
then do instead retrieve (EMP-OID = EMP.OID, EMP.name, EMP.age, EMP.salary)
where EMP.dept = ‘‘toy’’
```

and an incoming query:

```
retrieve (TOY_EMP.salary) where TOY_EMP.name = "Sam"
```

Clearly, utilizing the tuple-level rules system will entail materializing all the tuples in TOY_EMP in order to find Sam’s salary. It would be much more efficient to **rewrite** the query to:

```
retrieve (EMP.salary) where EMP.name = "Sam" and EMP.dept = "toy"
```

This section presents a general purpose rewrite algorithm for PRS2.

This second implementation is a module between the parser and the query optimizer and processes an incoming POSTGRES command, Q to which a rule, R applies by converting Q into an alternate form, Q’ prior to optimization and execution. This implementation can process any PRS2 rule which has a single command as its action, and we sketch the algorithm that is performed in this section. First, we present the algorithm for rules which have no event qualification and perform a concurrent example. Then we generalize the algorithm to rules with event qualifications.

Consider the rule:

```
define rule example_11
on replace to TOY_EMP.salary
then do instead replace EMP (salary = NEW.salary)
where EMP.name = CURRENT.name
```

and the incoming command:

```
replace TOY_EMP( salary = 1000) where TOY_EMP.name = "Joe"
```

Intuitively, we must remove any references to CURRENT.attribute and NEW.attribute in order to implement rules at a higher level than at individual tuple access. The first two steps of the algorithm remove these constructs. The third step deals with the semantics of POSTQUEL qualifications, while the final step perform semantic simplification if possible. The four steps to the algorithm now follow.

The first step is to note that CURRENT in the rule definition is really a **tuple variable** which ranges over the qualification in the user command. Hence, in R we must replace any reference to CURRENT.attribute with t-variable.attribute found as follows. If Q is a retrieve command, then t-variable.attribute is found in the target list. On the other hand, if Q is a replace or delete, then t-variable is the tuple variable being updated or deleted. In addition, the entire qualification from the user command must be added to the action part of the rule.

For the current example CURRENT will range over:

```
TOY_EMP where TOY_EMP.name = "Joe"
```

Hence, the rule can be converted in step 1 to:

```
on replace to TOY_EMP.salary
then do instead replace EMP (salary = NEW.salary)
where EMP.name = TOY_EMP.name and TOY_EMP.name = "Joe"
```

In step 2 of the algorithm we replace any reference to NEW.field-name with the right hand side of the target-list entry for the appropriate field name in the user's append or replace command.

In our example, we note that NEW.salary can be replaced by the constant "1000" and the rule is thereby further rewritten to:

```
on replace to TOY_EMP.salary
then do instead replace EMP (salary = 1000)
where EMP.name = TOY_EMP.name and TOY_EMP.name = "Joe"
```

The resulting action part of the rule now contains one or more tuple variables, in the above example TOY_EMP and EMP. Any update in POSTQUEL is actually a retrieve to isolate the tuples to be added, changed or deleted followed by lower level processing. Retrieves, of course, only perform a retrieval. As a result, step 3 of the algorithm must be executed for a tuple variable, t-variable if there exists a rule of the form:

on retrieve to t-variable.attribute
then do instead retrieve (attribute = *expression*) where QUAL

In this case, replace all occurrences of t-variable.attribute in R with *expression* and then add QUAL to the action part of the rule.

Such a rule exists as example_8, i.e:

```
define rule example_8
on retrieve to TOY_EMP
then do instead retrieve (EMP-OID = EMP.OID, EMP.name, EMP.age, EMP.salary)
```

```
where EMP.dept = ‘‘toy’’
```

Applying the rule in example_8 to further rewrite R we get:

```
on replace to TOY_EMP.salary
then do instead replace EMP (salary = 1000) from E1 in EMP
where EMP.name = E1.name and E1.name = "Joe" and E1.dept = "toy"
```

In the above example, we were required to rename the tuple variable to avoid a name conflict.

The last step of the algorithm is to notice that a semantic simplification can be performed to simplify the above rule to:

```
on replace to TOY_EMP.salary
then do instead replace EMP (salary = 1000)
where EMP.name = "Joe" and EMP.dept = "toy"
```

The resulting rule is now executed, i.e. the action part of the rule is passed to the query optimizer and executor. First, however, the query rewrite module must ensure that no new rules apply to the query which is about to be run. If so, steps 1-4 must be repeated for the new rule.

Intuitively, the user command has been substituted into the rule to accomplish the rewriting task in the four steps above. If the tag "instead" is present in R, then PRS will simply do the action part of the rewritten rule in place of Q. On the other hand, if "instead" is absent, then the action will be executed in addition to Q.

The above algorithm works if R has no event qualification. On the other hand, suppose R contains a qualification, QUAL. In this case before the above query modification phase we must transform the user command Q with qualification USER-QUAL into two user commands with the following qualifications:

USER-QUAL and **not** QUAL

USER-QUAL and QUAL

The first command is processed normally with no modifications while the second part has the algorithm above performed for it. Hence, two commands will be actually be run.

For example, consider the following rule:

```
define rule example_12
on retrieve to TOY_EMP where TOY_EMP.age < 40
then do instead retrieve (EMP-OID = EMP.OID, EMP.name, EMP.age, EMP.salary)
where EMP.dept = ‘‘toy’’
```

and the user query

```
retrieve (TOY_EMP.salary) where TOY_EMP.name = "Joe"
```

In this case, the following two queries will be run:

```
retrieve (TOY_EMP.salary) where TOY_EMP.name "Joe" and not TOY_EMP.age < 40

retrieve (TOY_EMP.salary) where TOY_EMP.name = "Joe" and TOY_EMP.age < 40
```

The first query is run normally while the second is converted by the algorithm to:

```
retrieve (EMP.salary) where EMP.name = "Joe" and EMP.dept = "Toy" and EMP.age < 40
```

As will be seen in Section 5, this corresponds to a relation that is partly materialized and partly specified as a view definition.

Rules for the query rewrite system must be supported by *Event* locks which are set at the relation level in the system catalogs. Locks set at the tuple level are not seen until until the command is in execution, and therefore the query rewrite module cannot use them.

4. CACHING AND CHOICE OF IMPLEMENTATION

For each rule, PRS2 must decide whether to use the query rewrite or tuple level implementation. Intuitively, if the event covers most of a relation, then the rewrite implementation will probably be more efficient while if only a small number of tuple are involved, then the tuple level system may be preferred. Specifically, if there is no where clause in the rule event, then query rewrite will always be the preferred option. The single rewritten query will be more efficient than the original users query with one or more separately evaluated action statements. On the other hand, if a where clause is present, then query rewrite will construct two queries as noted in the previous section. If a small number of tuples are covered by the event where clause, then these two queries will generally be less efficient than the single original query plus a separately activated action.

Hence, the choice of implementation should be based on the expected number of tuples that are covered by the event where clause. If this number, readily available from the query optimizer, is less than a cutoff value, CUTOFF-2, then the tuple level implementation should be chosen; otherwise the rewrite system should be used. Notice that CUTOFF-2 must be greater than or equal to CUTOFF-1 noted in the previous section. Further investigation may determine that the two numbers are the same.

However, the above discussion must be modified when caching of rules is considered, and we now turn to this subject. For any rule of the form:

```
on retrieve to object ...
then do instead retrieve ...
```

the action statement(s) can be evaluated in advance of rule activation by some user as long as the retrieve command(s) in the action part of the rule do not contain a function (such as user or time) which cannot be evaluated before activation. We call this advance activation **caching** the rule and there are 3 cases of interest:

1) The object noted in the rule event is a relation name and there is no where clause in the event.

In this case, the rule defines a complete relation and the query rewrite implementation would normally process the rule. If the rule is cached, then the query rewrite system should not convert the form of the query; rather it should simply execute the original user’s command on the cached data.

2) The object is of the form relation-name.field and there is no where clause in the event.

In this case, the rule defines a complete column of a relation. There is no query processing advantage to caching the complete column as a separate relation because the executor will have to perform a join between the relation of interest and the cached values to assemble the complete relation.

Rather, caching should be done on a tuple-by-tuple basis. Hence, the value of the action portion of the rule should be constructed for each tuple in the relation individually. This value can either be stored directly in the tuple or it can be stored in a convenient separate place. If it is stored in the tuple, then the same value may be materialized several times for various tuples. On the other hand, if it is stored separately, then the executor must maintain a pointer from the tuple to the cached value and pay a disk seek to obtain the value. The tradeoffs between these two implementations have been studied in [JHIN88], and we expect to implement both tactics.

If the rule is cached on a tuple-by-tuple basis, then query rewrite must not be performed. Rather, execution should proceed in the normal fashion, and the executor will simply access the cache to obtain the value needed. On the other hand, if a particular value is not cached for some reason, the executor must evaluate the rule for the tuple of interest to get the needed value.

Consequently, if a rule which defines a whole column is cached, then the query rewrite implementation, which would normally process the rule, should do nothing and let the executor fetch values from the cache assisted by the tuple level implementation if needed values are not present in the cache.

3) There is a where qualification.

In this case, the action part applies to (perhaps) many tuples, and we must cache the action part for each tuple to which it applies. This cache can be either in the tuple itself or in some separate place. Hence, it behaves exactly like case 2) above.

Whenever the action part of the rule is cached, "invalidation" locks must be set on each accessed field in each tuple. If any other user command updates a field on which there is an invalidation lock, then the cached value must be invalidated before proceeding; however, invalidation locks are left in place. Thus, any object that has been cached once has its locks in place permanently.

POSTGRES will decide what rules to cache, and expects to use the following algorithm. The general idea is to keep the "most worthy" objects in the cache, i.e. those which benefit the most from a cached representation. The benefit computation uses the following parameters for the ith object:

S_i: the size of the cached object
M_i: the cost to materialize the object
A_i: the cost to access the object if cached
D_i: the cost to invalidate the object if cached
a_i: the frequency of accesses to this object
u_i: the frequency of updates (invalidations) to this object

In this case, the expected cost per unit time, C_i, for object i is:

$$C_i = \begin{cases} a_iA_i + u_iD_i & \text{if i is cached} \\ a_iM_i & \text{if i is not cached} \end{cases}$$

Using an indicator variable I_i to indicate whether i is cached, the above can be expressed as:

$$C_i = a_iM_i - [a_i(M_i - A_i) - u_iD_i]I_i$$

Thus, the total expected cost per unit time is minimized when

$$F = \sum_{i=1}^{N}[a_i(M_i - A_i) - u_iD_i]I_i$$

is maximized. The constraints on this optimization problem are:

$$\sum_{i=1}^{N} S_iI_i \le S$$

where S is the size of the cache. It can be shown that the solution of the above 0-1 Knapsack problem can be approximated by the following algorithm if $\mathrm{Max}(S_i) \ll S$:

Define the benefit function f_i as

$$f_i = \frac{a_i(M_i - A_i) - u_iD_i}{S_i}$$

Intuitively, the benefit function is the benefit per unit size of caching the ith object. Now sort the objects in the descending order of their benefit functions. Let the j^{th} entry in this order be the object π_j. Now define:

$$B_{min} = f_{\pi_k} \qquad \text{where } \sum_{j=1}^{k} S_{\pi_j} \le S \text{ and } \sum_{j=1}^{k+1} S_{\pi_j} > S$$

Then,

$$I_i = \begin{cases} 1 & \text{if } f_i \ge B_{min} \\ 0 & \text{otherwise} \end{cases}$$

Thus at any stage we keep only those objects in cache whose benefit function exceeds the current B_{min}.

The only issues that remain are to choose B_{min} and to efficiently estimate a_i and u_i. We describe below a unique approach to each of these problems.

Consider the access pattern for an object i. Let L_a be the expected length of the interval I_a defined to be the time between the first access following an update to i and the first update to the object following this access. It can be shown that $L_a = \frac{1}{u_i}$. Thus, if we measure the average length of all the I_a intervals seen, we have a good statistical estimate for u_i. It can be seen that the average length of a similar interval determines a_i.

These average length statistics are very cheap to maintain for cached objects because POSTGRES must invalidate the cached object on update and then rebuild the cache on the first subsequent access. Both actions require writing the cached object. Hence, if the statistics are kept with the cached object, there is no extra I/O cost to maintain them.

For each object whose current benefit exceeds B_{min} we maintain these statistics. Periodically, a caching demon recomputes the current benefit of all objects whose statistics are being maintained and deletes those objects from the cache (and stops maintaining their statistics) whose benefit is below the current B_{min}.

In between runs of the daemon, a cached object is marked "invalid" on each update. On an access, an object is materialized and cached if

1) Its statistics are being maintained and it is currently "invalid", or

2) Its statistics are not being maintained, but its **expected** benefit exceeds B_{min}. In this case, we also start keeping its statistics. The expected benefit of an object i is defined to be:

$$\bar{f}_i = \frac{\bar{a}(M_i - A_i) - \bar{u}D_i}{S_i}$$

where $\bar{a}$ is the average frequency of access to the objects in the database, and $\bar{u}$ is the average update frequency in the database. Ideally we would like to keep only those objects whose **actual** benefit exceeds B_{min}. Since the past access patterns of these objects are not being kept, the best we can do is to make the decision on the basis of expected benefit.

When the cache demon runs, it checks for the space utilization. If the cache space is not entirely utilized, it increases the value of B_{min}. It thus deletes fewer objects, and also permits the caching of more new objects in between runs. On the other hand, if the cache space is used up in between runs, then the next time the daemon runs, it will use a lower value of B_{min}. This feedback mechanism ensures that B_{min} will converge to the correct value, provided the database access patterns are fairly stable. Furthermore, if access patterns change dramatically, this feedback mechanism will ensure a reconvergence to a new B_{min}.

5. SUPPORT FOR VIEWS

5.1. Normal Views

In POSTGRES named procedures can be defined to the system as follows:

```
define [updated] procedure proc-name (type-1, ..., type-n) as postquel-commands
```

If the procedure has no parameters, then it is a **view** definition, and a syntactic alternative to the above definition would be:

```
define [updated] view view-name as postquel-commands
```

For example, consider the following view definition:

```
define view SHOE_EMP as
retrieve (EMP.all) where EMP.dept = “shoe”
```

This definition would be turned automatically by POSTGRES into the following rule:

```
define rule SHOE_EMP_ret
```

```
on retrieve to SHOE_EMP
then instead do retrieve (EMP-OID = EMP.OID, EMP.all) where EMP.dept = ‘‘shoe’’
```

This rule will be processed by the query rewrite implementation and will perform the correct query modification to any user query that appears.

On the other hand, consider the following view definition:

```
define updated view TOY_EMP as
retrieve (EMP.name, EMP.age, EMP.salary) where EMP.dept = ‘‘toy’’
```

In this case, the user wishes standard view update semantics to be applied to user updates to TOY_EMP. Hence the system would automatically construct the retrieve rule in example_8 in addition to the following update rules:

```
define rule TOY_EMP_d
on delete to TOY_EMP
then do instead delete EMP where EMP.OID = CURRENT.EMP-OID

define rule TOY_EMP_a
on append to TOY_EMP
then do instead append to EMP (name = NEW.name,
                        age = NEW.age,
                        salary = NEW.salary,
                        dept = ‘‘toy’’)

define rule TOY_EMP_r
on replace to TOY_EMP
then do instead replace EMP (name = NEW.name, age = NEW.age, salary = NEW.salary)
                             where EMP.OID = CURRENT.EMP-OID
```

A view will automatically have an OID field for each tuple variable in its definition. When multiple identifiers are present, the name of each identifier is tvar-OID, where tvar is the tuple variable involved. This identifier keeps the OID of a tuple in the relation tvar which was used to construct this particular tuple in the view.

If a view is not specified as updated, then it is the responsibility of the user to specify his own update rules as discussed in the next subsection.

5.2. More General Views

Consider a rule defining a conventional join view, e.g:

```
define rule example_13
on retrieve to E-D
then do instead
retrieve (EMP-OID = EMP.OID, EMP.name, EMP.salary, EMP.dept,
DEPT-OID = DEPT.OID, DEPT.floor)
where EMP.dept = DEPT.dname
```

In current commercial systems all deletes and appends fail for the E-D view and some updates fail. However, in PRS2 we can specify the following collection of update rules:

```
define rule E-D-1
on replace to E-D.name, E-D.salary
then do instead replace EMP (name = NEW.name, salary = NEW.salary)
where EMP.OID = CURRENT.EMP-OID

define rule E-D-2
on replace to E-D.floor
```

```
then do instead replace DEPT (floor = NEW.floor)
where DEPT.OID = CURRENT.DEPT-OID

define rule E-D-3
on replace to E-D.dept
then do instead
        replace EMP (dept = NEW.dept)
        where EMP.OID = CURRENT.EMP-OID

        append to DEPT (dname = NEW.dept, floor = CURRENT.floor)
        where NEW.dept not-in {DEPT.dname}
```

This will map all updates to underlying base relations. Moreover, when an employee in the E-D view changes departments, then his new department is created if it doesn't exist. All other reasonable rules for updating EMP and DEPT when updates to E-D occur appear expressible as PRS2 rules. Hence, a sophisticated user can specify any particular mapping rules for updates that he wishes to. In this way, **all** views can be made updatable, a big advance over current implementations.

5.3. Materialized, Partial and Composite Views

In this section we discuss three other more general kinds of views that are possible with PRS2. First, consider a materialized view, e.g. one whose tuples are maintained automatically by the system. Such views are discussed in [BLAK86, ROUS89]. In PRS2, all views may be materialized by caching the action part of the rule that defines the view. This will be done automatically by the caching demon if the view is "worthy" as noted earlier. The only inefficiency in PRS2 is that materialized views are invalidated if an update to an underlying base relation occurs. In the future we propose to study how to incorporate algorithms to directly update materialized views.

Next consider a **partial** view, i.e a relation that is partly instantiated with tuples and partly expressed by a view. Hence, a relation is considered to be the union of a stored relation and a view. This is naturally supported by the following retrieve rule:

```
define rule example_14
on retrieve to PARTIAL
then do retrieve (EMP-OID = EMP.OID, EMP.all) where EMP.dept = ‘‘toy’’
```

Here, notice that the keyword "instead" has been omitted; consequently a retrieve to the stored relation PARTIAL will occur along with query rewrite to access the employees in the toy department. It is easy to specify updating rules for partial views, and example_15 expresses the appropriate insert rule:

```
define rule example_15
on append to PARTIAL
then do instead
  append to EMP (NEW.all) where NEW.dept = ‘‘toy’’
  append to PARTIAL (NEW.all) where NEW.dept != ‘‘toy’’
```

Lastly, consider a view which is a composite of two relations, e.g:

```
define view TOY_EMP as
retrieve (EMP.name, EMP.salary, EMP.age) where EMP.dept = ‘‘toy’’
retrieve (NEWEMP.name, NEWEMP.status, NEMEMP.age)
where NEWEMP.dept = ‘‘toy’’
```

This can easily be expressed as:

```
define rule example_16
on retrieve to TOY_EMP
then do instead
retrieve (EMP-OID = EMP.OID, EMP.name, EMP.salary, EMP.age) where EMP.dept = ‘‘toy’’
retrieve (NEWEMP-OID = NEWEMP.OID, NEWEMP.name, NEWEMP.status, NEMEMP.age)
where NEWEMP.dept = ‘‘toy’’
```

Clearly various composite views can be defined, with automatic or user defined updating rules.

6. PROCEDURAL FIELDS

There are four ways that procedures exist in the POSTGRES system. Procedures without parameters are effectively view definitions and were discussed in the previous section. In this section we discuss procedures with parameters, general procedural fields and special procedural fields.

6.1. Procedures with Parameters

Procedures can be defined which contain parameters, e.g:

```
define procedure TP1 (char16, int4) as
replace ACCOUNT (balance = ACCOUNT.balance - $2) where ACCOUNT.name = $1
```

This procedure is not a view definition and is simply **registered** in the system catalogs. In this case, the only functionality supported by POSTGRES is executing the procedure, e.g:

```
execute TP1 ("Sam", 100)
```

6.2. General Procedural Fields

General procedures have been proposed for POSTGRES as a powerful modelling construct. In this section we demonstrate the general procedures are efficiently simulated by PRS2. Consider the standard example from [STON87]:

```
EMP (name, hobbies)
```

Each tuple of EMP contains a procedure in the hobbies field. Since there is a command to define procedures, each row of EMP has a value which is a registered procedure of the form:

```
proc-name(param-list)
```

where proc-name is a previously registered procedure. In this case an insert would look like:

```
append to EMP (name= "Sam", hobbies = foobar(param-1, ..., param-n))
```

The parameter list will be present only if the registered procedure has parameters.

This data type can be effectively supported by defining one rule per procedural data element of the form:

```
on retrieve to rel-name.column-name where rel-name.OID = value
then do instead execute proc-name(param-list)
```

Hence, the insertion of Sam can be done as follows:

```
append to EMP (name = "Sam")

define rule example_17
on retrieve to EMP.hobbies where EMP.OID = value
then do instead execute foobar(param-1, ..., param-n)
```

If the action part of the rule is cached, then this will correspond to the caching of procedures discussed in [STON87].

Moreover, POSTGRES has syntax to update the data base through a procedural field. Consider for example the following update:

```
replace EMP.hobbies (position = ‘‘catcher’’) where EMP.name = ‘‘Sam’’
```

There are two situations of interest. First, if the procedure corresponding the Sam’s hobbies has no parameters, then it could have been specified as a view definition with automatic update. In this case, the automatic update rules can be applied to map the above update. Otherwise, the user is free to add his own updating rules for Sam’s hobby field, e.g:

```
define rule example_18
on replace to EMP.hobbies where EMP.name = "Sam"
then do instead ....
```

6.3. Special Procedures

A relation in POSTGRES can have a field of type special procedure. Consider the DEPT relation from Section 1:

```
create DEPT (dname = char16, floor = i4, composition = EMPS)
```

Here, composition is intended to have a value found by executing the procedure EMPS. This procedure would be defined as:

```
define procedure EMPS (char16) as
retrieve (EMP-OID = EMP.OID, EMP.all) where EMP.dept = $.dname
```

Like TP1, this procedure contains a parameter, $.dname, and therefore is not a view definition. Like TP1 it is merely registered at the time it is defined.

At the time the DEPT relation is created, the following rule can be defined:

```
on retrieve to DEPT.composition
then do instead retrieve (EMP-OID = EMP.OID, EMP.all)
where EMP.dept = CURRENT.dname
```

This rule defines the column, DEPT.composition, and query rewriting rules will map any query containing a reference to composition correctly. Moreover, caching can be applied to this rule and the tuples for each given department name will be optionally cached by the algorithms in Section 4.

7. CONCLUSIONS

In this paper we have demonstrated a rule system that will take appropriate actions when specific events become true. This rules system can have CURRENT.attribute and NEW.attribute in the action part, and its semantics are naturally defined when individual tuples are retrieved or modified. We described the corresponding tuple level implementation which enforces these semantics. However, we also presented an algorithm through which we can remove CURRENT and NEW from a rule and perform query rewrite to enforce the rule, and described this second implementation of the rules system.

Moreover, any rule whose event and action parts are retrieves can be evaluated before it is activated by a user. We described how this caching takes place and what algorithm should be used to manage the cache.

Then, we demonstrated that support for relational views is merely an application of our query rewrite implementation on a rule which specifies the view definition. Moreover, non standard update semantics can be specified as additional updating rules, thereby substantially enchancing the power of views. Various more general views can also be readily supported.

We then showed that POSTGRES procedures are merely an additional application of the rules system. In addition, caching of rules naturally supports materialized views and cached procedures, thereby no extra mechanisms are required to obtain this functionality.

In the future we are going to explore additional applications of PRS2. These include the possibility of writing a physical data base design tool and the POSTGRES version system in PRS2. Moreover, we plan to search for algorithms which could convert an early implementation of a rule, e.g:

```
on replace ...
then do replace ...
```

to an equivalent late rule, e.g:

```
on retrieve ...
then do instead retrieve ...
```

This would allow PRS2 to cache the action part of the rule and decide automatically between early and late evaluation.

Lastly, we will continue to support our earlier **always** and **never** syntax, because it can be easily compiled into multiple PRS2 commands. We also plan to explore other higher level interfaces, for which rule compilers can be constructed.

REFERENCES

[BLAK86] Blakeley, J. et. al., ‘‘Efficiently Updating Materialized Views,’’ Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., June 1986.

[CODD74] Codd, E., ‘‘Recent Investigations in Relational Data Base Systems,’’ IBM Research, Technical Report RJ1385, San Jose, Ca., April 1974.

[DADA86] Dadam, P et. al., ‘‘A DBMS Prototype to Support Extended NF2 Relations: An Integrated View on Flat Tables and Hierarchies,’’ Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., June 1986.

[DELC88] Delcambre, L. and Etheredge, J., “The Relational Production Language,” Proc. 2nd International Conference on Expert Database Systems, Washington, D.C., February 1988.

[ESWA76] Eswaren, K., ‘‘Specification, Implementation and Interactions of a Rule Subsystem in an Integrated Database System,’’ IBM Research, San Jose, Ca., Research Report RJ1820, August 1976.

[HANS89] Hanson, E., ‘‘An Initial Report on the Design of Ariel, ’’ ACM SIGMOD Record, Sept. 1989.

[JHIN88] Jhingran, A., ‘‘A Performance Study of Query Optimization Algorithms on a Data Base System Supporting Procedural Objects,’’ Proc. 1988 VLDB Conference, Los Angeles, Ca., Sept. 1988.

[KOLO89] Kolovson, C. and Stonebraker, M., ‘‘Segmented Search Trees and their Application to Data Bases,’’ (in preparation).

[MCCA89] McCarthy, D. and Dayal, U., ‘‘The Architecture of an Active Data Base Management System,’’ Proc 1989 ACM-SIGMOD Conference on Management of Data, Portland, Ore., June 1989.

[ROUS87] Rousoupoulis, N., ‘‘The Incremental Access Method of View Cache: Concepts, Algorithms, and Cost Analysis,’’ Computer Science Technical Report CS-TR-2193, University of Maryland, February 1989.

[ROWE87] Rowe, L. and Stonebraker, M., “The POSTGRES Data Model,” Proc. 1987 VLDB Conference, Brighton, England, Sept 1987.

[STON75] Stonebraker, M., ‘‘Implementation of Integrity Constraints and Views by Query Modification,’’ Proc. 1975 ACM-SIGMOD Conference, San Jose, Ca., May 1975.

[STON82] Stonebraker, M. et. al., ‘‘A Rules System for a Relational Data Base Management System,’’ Proc. 2nd International Conference on Databases,’’ Jerusalem, Israel, June 1982 (available from Academic press).

[STON86] Stonebraker, M. and Rowe, L., “The Design of POSTGRES,” Proc. 1986 ACM-SIGMOD Conference, Washington, D.C., June 1986.

[STON87] Stonebraker, M., et. al., ‘‘Extending a Data Base System With Procedures,’’ ACM TODS, September, 1987.

[STON88] Stonebraker, M. et. al., “The POSTGRES Rules System,” IEEE Transactions on Software Engineering, July 1988.

[WIDO89] Widom, J. and Finkelstein, S., ‘‘A Syntax and Semantics for Set-oriented Production Rules in Relational Data Bases, IBM Research, San Jose, Ca., June 1989.