A COMMENTARY ON THE POSTGRES RULES SYSTEM

Michael Stonebraker, Marti Hearst, and Spyros Potamianos
EECS Dept.
University of California, Berkeley

Abstract

This paper suggests modifications to the POSTGRES rules system (PRS) to increase its usability and function. Specifically, we suggest changing the rule syntax to a more powerful one and propose additional keywords, introduce the notion of rulesets whose purpose is to increase the user's control over the rule activation process, and expand the versioning facility to support a broader range of applications than is currently possible.

1. INTRODUCTION

In the POSTGRES data base management system [WENS88, STON86a], we have designed and implemented an integrated rules system. Based on this experience we have several changes we plan to make. This paper reviews (in Section 2) the initial proposal as presented in [STON88], describes the status of the current implementation, and suggests modifications and additions in three areas: rule syntax (Section 3), rulesets (Section 4), and versions (Section 5). In Section 6 we summarize the changes.

2. THE CURRENT POSTGRES RULES SYSTEM

2.1. Syntax

POSTGRES supports a query language, POSTQUEL, which borrows heavily from its predecessor, QUEL [HELD75]. The main extensions are syntax to deal with procedural data, extended data types, rules, inheritance, versions and time. The language is described elsewhere [ROWE87], and here we give only one example to motivate our rules system. The following POSTQUEL command sets the salary of Mike to the salary of Bill using the standard EMP relation:

```
replace EMP (salary = E.salary)
using E in EMP
where EMP.name = ''Mike''
and E.name = ''Bill''
```

POSTGRES allows any such POSTQUEL command to be tagged with three special modifiers which change its meaning. Such tagged commands become **rules** and can be used in a variety of situations as will be presently described.

The first tag is ''always'' which is shown below modifying the above POSTQUEL command.

```
always replace EMP(salary = E.salary)
using E in EMP
where EMP.name = ''Mike''
and E.name = ''Bill''
```

The semantics of this rule are that the associated command should logically appear to run forever. Hence, POSTGRES must ensure that any user who retrieves the salary of Mike will see a value equal to that of Bill's.

If a retrieve command is tagged with ''always'' it becomes a rule which functions as an **alerter.** For example, the following command will retrieve Mike's salary whenever it changes.

```
always retrieve (EMP.salary)
where EMP.name = ''Mike''
```

This research was sponsored by the Army Research Organization Grant DAAL03-87-0083 and by the Defense Advanced Research Projects Agency through NASA Grant NAG 2-530.

The second tag which can be applied to any POSTQUEL command is ‘‘refuse’’. This tag is useful to enforce sophisticated protection rules as noted in [STON88]. The final tag is ‘‘one-time’’ which is useful to implement ‘‘alarm clocks’’, i.e. rules which fire once and then permanently disappear.

2.2. Implementation Options

Currently the PRS optimizes rule execution along two dimensions. As an example of the first, consider the following collection of rules:

```
always replace EMP(salary = E.salary)
using E in EMP
where EMP.name = ‘‘Mike’’
and E.name = ‘‘Bill’’

always replace EMP(salary = E.salary)
using E in EMP
where EMP.name = ‘‘Bill’’
and E.name = ‘‘Fred’’
```

These rules ensure that Mike’s salary is set to Bill’s which is set to Fred’s. If the salary of Fred is changed, then the second rule can be awakened to change the salary of Bill which can be followed by the first rule to alter the salary of Mike. In this case an update to the data base awakens a collection of rules which in turn awaken a subsequent collection. This control structure is known as **forward chaining,** and we will term it **early** evaluation. The first option available to the PRS is to perform early evaluation of rules which results in a forward chaining control flow.

A second option is to delay the awakening of either of the above rules until a user requests the salary of Bill or Mike. Hence, neither rule will be run when Fred’s salary is changed. Rather, if a user requests Bill’s salary, then the second rule must be run to produce it on demand. Similarly, if Mike’s salary is requested, then the first rule is run to determine Mike’s salary, in turn triggering the second rule which is run to obtain Bill’s salary. This control structure is known as **backward chaining,** and we will term it **late** evaluation. The choice of early or late evaluation is an internal optimization subject to a collection of restrictions presented in [STON88].

There is another dimension to PRS optimization which deals with the rule firing mechanism. In [STON86b] we analyzed the performance of a rule indexing structure and various structures based on physical marking (locking) of objects to control rule activation. When the average number of rules that covered a particular tuple was low, locking was preferred. Moreover, rule indexing could not be easily extended to handle rules with join terms in the qualification. Because we expect there will be a small number of rules which cover each tuple in practical applications, we are utilizing a locking scheme.

Consequently, when a rule is installed into the data base for either early or late evaluation, POSTGRES runs the command corresponding to the rule in a special mode and collects a list of all data items that are read or proposed for writing by the rule. On each such data item the system sets one of several kinds of locks, detailed in [STON88]. When a query subsequently reads or writes one of these marked objects, the locks trigger appropriate rule specific processing. This processing is termed ‘‘record processing’’ because it is activated by updates, insertions or deletions of individual records.

There are situations where lock escalation may be desirable. For example, consider the rule:

```
always replace EMP ( desk = ‘‘steel’’)
where EMP.age < 80
```

Because this rule will read the ages of most employees, it is preferable to **escalate** individual locks on data items to a column level lock. In this case, rule activation can be performed by **rewriting** the user interaction. For example, the query

```
retrieve (EMP.desk)
where EMP.name = ‘‘Mike’’
```

can be easily rewritten as:

```
retrieve (‘‘steel’’)
where EMP.name = ‘‘Mike’’
and EMP.age < 80

retrieve (EMP.desk)
where EMP.name = ‘‘Mike’’
and EMP.age >= 80
```

The complete collection of PRS rewrite rules is specified in [STON88]. In summary, the PRS must choose for any rule whether to use early or late evaluation and whether to use record processing or query rewriting as the rule processing algorithm. The tradeoffs between these options are also considered in [STON88].

2.3. Current Implementation Status

The current PRS implementation supports only late evaluation for always replace rules. In addition, both fine and coarse granularity locks are working. However, we have only implemented the record processing algorithms, so coarse granularity locks are used only to indicate that record level processing must be done for each record in the column in question. Consequently, the escalation of locks does not change the fundamental rule processing algorithm, but merely saves space.

3. SYNTAX CHANGES

Based on our initial experience with the rules system and feedback from early candidate users, we are considering modifying the rules system in the ways discussed in this and the next two sections.

3.1. Problems with the Current Syntax

There are at least three problems with the current syntax. First, it is impossible to specify transition constraints, such as permitting salary adjustments only if they are less than $500. This would require something like:

refuse replace EMP (salary)
where new.salary - old.salary > 500

In the current PRS there is no way to refer to the new value or the old value of a field. Moreover, simply adding new and old as keywords changes the semantics of PRS. For example, the above command only makes sense when there is a new value for some employee's salary. This is not consistent with the current paradigm of a rule always being in execution.

Second, there is insufficient control to implement all useful cases of referential integrity [DATE81]. Consider for example the EMP and DEPT relations:

EMP (name, salary, dept)
DEPT (dname, floor).

and suppose one wanted to delete all employees in a department as a side effect of deleting the department. This is easily specified as the following PRS rule:

delete always EMP
where EMP.dept not-in {DEPT.dname}

However, the above rule also refuses the insertion of employees in a non-existent department. Hence, it implements rule ''cascade the deletion and refuse the insertion''. Unfortunately, there is no way to cascade deletions but take some other action on insertions. Hence, there is insufficient granularity of control.

Lastly, although the PRS is a powerful rules system, it has always been frustrating to some of us that it could not be used to support view processing for relational views. Hence, the function provided by PRS seems a slight mismatch with the needs of a DBMS rules system. This concern has caused us to consider changing the rules system to the following one.

3.2. PRS II

PRS contains a rules system that performs either tuple processing or query rewrite, depending on the lock granularity, and automatically determined which rules to evaluate early and which late. This uniform specification of rules with different system-determined implementations will be modified. Specifically, to add additional control we are changing the syntax slightly and adding keywords **new** and **old**. This changes the paradigm from the notion of a command perpetually in execution to one where events are specified which cause specific actions to occur. Unfortunately this will require us to abandon optimizing the early versus late decision in most cases. It also makes PRS II closer to other proposals, e.g. [DELC88, DAYA86].

The syntax of a PRS II rule is thereby:

define rule rulename **is**
on POSTQUEL event
do POSTQUEL command(s)

The semantic interpretation is that the action part of the rule is executed once when the POSTQUEL event occurs. Multiple rules may be defined based on the same event; all applicable ones are executed. POSTQUEL events are specified using the following syntax:

on command-name to object
where condition

Here **command-name** is one of {append, replace, delete, retrieve} optionally coupled with **new** on an append or replace and **old** on a replace or delete. **Object** is the name of a relation or a column in a relation and **condition** is an arbitrary POSTQUEL qualification. **Command(s)** are any set of legal POSTQUEL commands with two

extensions. First, a POSTQUEL **refuse** command with a target list of columns which cannot be modified is provided. This command allows a rule to refuse the update that its event portion specifies. Second, we add the keywords **new** and **old** which can be used anywhere that a relation name, tuple variable or constant can. The **new** keyword refers to the tuple being inserted in an event involving an append command or the tuple being updated in a replace event. The **old** keyword refers to the tuple that is being removed by a delete event or the one to be updated by a replace event.

Consider the following rule, using the new syntax, that propagates Bill's salary on to Mike:

```
on replace to EMP.salary
where EMP.name = ''Bill''
do replace EMP (salary = new.salary)
where EMP.name = ''Mike''
```

In this rule, the event specified by the **on** condition is the introduction of a new value for the salary of the employee named Bill. At the time this rule is awakened, there will be a **new** tuple and there may or may not be an **old** tuple with the fields that are being replaced. When the rule manager is awakened, it processes the rule by running the query specified by the **do** statement, after first substituting values for fields specified by new.column-name or old.column-name.

However, the above rule does not preclude a user from directly updating Mike's salary. Hence, by itself, it is not equivalent to the PRS rule discussed in the previous section. To attain equivalence, we must add a second rule:

```
on replace to EMP.salary
where EMP.name = ''Mike''
do refuse (new.salary)
```

The second rule is assigned a lower priority than the first, and ensures that no other update can change Mike's salary.

Detecting an append, delete or replace that satisfies the on-condition and then executing the do-statements corresponds to normal forward chaining. In fact, any PRS append, delete or replace statement used with early evaluation can be routinely converted into equivalent PRS II rules. The PRS "always retrieve" rules are also easily translated. For example, the following PRS II rule returns Mike's salary each time it changes:

```
on replace to EMP.salary
where EMP.name = ''Mike''
do retrieve (new.salary)
```

Backward chaining rules are also easily expressed. The following rule expresses the stipulation that Mike should earn the same salary as Bill:

```
on retrieve to EMP.salary
where EMP.name = ''Mike''
do retrieve (EMP.salary)
where EMP.name = ''Bill''
```

This rule is awakened when a query attempts to read Mike's salary. At that time the do-action is run and retrieves the current salary of Bill instead. Of course, Bill's salary could be specified in a similar way and a backward chaining control flow results.

If a PRS II rule contains **new** or **old**, then record processing will be used to implement the rule. However, PRS II rules which do not contain these keywords will be implemented via query rewrite. For example, consider the rule:

```
on retrieve to EMP.desk
retrieve ''steel'' where EMP.age < 80
```

This rule can use a column level lock and be easily enforced by a query rewrite algorithm, essentially the same as the one in the example in the previous section and in [STON88].

In summary, PRS II can be supported by the same implementation used in PRS; that of item locks or column locks. Item locks are supported by tuple processing while column locks are supported by query rewrite. The decision on lock granularity can be made automatically only for rules which do not use the keywords **new** and **old**. The way rules are written determines whether they will be executed in a forward or backward chaining manner, as seen in the two salary writing examples above. Backward chaining rules can be **precomputed** by a demon and their results cached. However, forward chaining rules cannot be delayed to late evaluation without changing their semantics. Hence, there is less room for automatic optimization in PRS II. On the other hand, PRS II has augmented power as demonstrated in the next section.

3.3. New Functionality

In this section we indicate how to use PRS II to perform transition constraints and solve the view update problem. Restricting salary updates to $500 is easily accomplished with the following rule:

```
on new EMP.salary
refuse (new.salary)
where new.salary - old.salary > 500
```

The other example concerns view update. Assume that we have the following view definition for a conventional relational DBMS:

```
define view
EMP-DEPT (EMP.all, DEPT.floor)
where EMP.dept = DEPT.dname
```

This view is easily expressed as the following PRS II rule:

```
on retrieve to EMP-DEPT
do retrieve (name = EMP.name,
salary = EMP.salary,
dept = EMP.dept,
floor = DEPT.floor)
where EMP.dept = DEPT.dname
```

Standard query modification [STON75] for retrieve commands is equivalent to the processing performed by query rewrite for PRS II. However, the problem with views is that many updates are semantically ambiguous. It is not clear, for example, how to process the following ambiguous update:

```
replace EMP-DEPT (floor = 1)
where EMP-DEPT.name = ‘‘Mike’’
```

In PRS II, we can allow a user to specify additional rules to control system behavior in these circumstances. Specifically, suppose that it is known that there is exactly one department on each floor. This means that the intent of the update above is to place Mike in the department located on the first floor. This can be enforced by the following rule:

```
on update to EMP-DEPT.floor
do replace EMP (dept = DEPT.dname)
where DEPT.floor = new.floor
and EMP.name = new.name
```

Processing of an update to EMP-DEPT occurs in two stages. First, a retrieval operation must be performed to isolate the records to be modified. This query is rewritten by the ‘‘on retrieve’’ rule to obtain the desired data. Next, the system computes proposed updates to EMP-DEPT. Each such proposed update will fire the ‘‘on update’’ rule, which will make the correct actual update.

Hence, views can be supported by **two** kinds of rules. The first kind specifies how to handle retrievals, and triggers standard query modification. The second kind of rules specifies what actions to take with proposed updates of individual tuples in the view. These rules can resolve ambiguity in an application specific way and are the component that is missing from current commercial implementations of views.

Of course, it is possible to have default update processing perform the standard algorithm when no update rules are specified by the user. In this way, no extra effort is required for view updates which are not ambiguous.

4. RULESETS

4.1. Introduction

Currently the rules defined within a POSTGRES data base are monolithic in organization, i.e. all rules are active for all users at all times. This is problematic for several reasons. First, a knowledge engineer frequently needs to modify active rules during the development phase. Currently, this must be done tediously by deleting and reinserting individual rules. Moreover, when multiple rules are inserted in the current system, since the rules become active immediately, synchronization hazards can occur. For example, consider the following two rules:

```
replace always EMP (salary = E.salary)
using E in EMP
where E.name = ‘‘Mike’’
and E.salary < 10000
and EMP.name = ‘‘Joe’’
priority = 2

replace always EMP (salary = E.salary)
using E in EMP
where E.name = ‘‘Jean’’
and E.salary > 10000
and EMP.name = ‘‘Mike’’
priority = 10
```

Suppose the rules are entered in the above order and Mike’s salary is initially $5000. In this case, the first rule will fire immediately and change Joe’s salary. On the other hand, if the second rule is entered first, then Mike’s salary will be adjusted to a number above $10,000. and Joe’s salary will not get changed. Such ordering dependencies should be avoided.

A third motivation for rulesets concerns applications which have a rule base composed of both shared and private rules. For example,

consider a POSTGRES implementation of a text retrieval system such as RUBRIC [MCCUN86]. Users specify a taxonomy which is used to classify an incoming stream of articles into subject areas. For example, if a user is interested in articles about scuba diving, he might divide the topic into subtopics such as "coral_reefs," "equipment_maintenance," "dangers," and divide these into subtopics as well, down to the level of lexical groupings. When an article is scanned, its goodness of fit is assessed for each leaf level subcategory. This result is propagated up the taxonomy and certainty of membership is computed for each category. When taxonomies are large, users will wish to use a predefined set of categorization rules with some private modifications for their own tastes. This combination of shared and private rules is awkward to specify in the current system.

As a remedy to these problems, we are introducing **rulesets** into POSTGRES. Rulesets may be defined and removed at will, and each POSTGRES rule may optionally be placed into a ruleset. Rulesets are hierarchically structured, and can be **activated** or **deactivated**.

4.2. Ruleset Commands

A ruleset is defined as follows:

Define Ruleset ruleset_name
[inherits ruleset {, ruleset }]
[init_script proc-name]
[cleanup_script proc-name]

It is convenient to group rulesets into an inheritance hierarchy, so that common collections of rules can be shared among multiple rulesets. The **inherits** clause provides this capability. The **init_script** is the name of a POSTGRES procedure which contains a script of POSTQUEL commands to be run when the ruleset is activated. Typically this script contains initialization information and instructions to create any necessary temporary relations. The **cleanup_script** is a procedure containing POSTQUEL queries that is invoked at deactivate time. A ruleset can be removed with the following command:

Remove Ruleset ruleset_name

Rules can be bound to an individual ruleset and subsequently removed from a ruleset using the following two commands:

Add rule-name to ruleset
Remove rule-name from ruleset

Rulesets can be activated using the following command:

Activate ruleset_name
[i_script]
[late_signal]
[auto_deactivate]

Here, the **i_script** flag, if set, indicates that the initialization script be run. Normally, an activate command is acknowledged immediately by the POSTQUEL run-time system. However, the user can optionally chose to be notified only when the inference engine detects that no new inferences have been made as a result of the ruleset activation by setting the **late_signal** flag. This allows assessment of the state of a cascaded computation at the conclusion of the inference process. Under normal circumstances, a ruleset remains active until deactivated. However, the user can specify with the **auto_deactivate** flag that ruleset deactivation should occur as soon as ‘‘the dust settles.’’

The last command is

Deactivate ruleset_name
[d_script]

This command sets all members of ruleset_name to "inactive" status. The **d_script** flag, when set, indicates that the deactivation script should be run.

4.3. Implementation

Each rule is included in a unique ruleset, and the ruleset name is stored with the command body of the rule. POSTGRES will maintain a main memory hash table to indicate which rulesets are active. If a rule is awakened by the rule manager, a check is initiated to ensure that it is in an active ruleset. If so, processing continues normally; otherwise the rule is ignored.

For space and implementation efficiency the lock information placed on individual tuples will not be updated with "active" markers. Hence, inactive rules will cause the rule engine to fetch the command body of the rule before realizing that the rule currently has no effect (although we may opt to cache rule/ruleset mapping information). The alternative is to have deactivation of rules be very expensive, requiring that all rule locks be found and updated.

5. AUGMENTED VERSION SUPPORT

In a rules system application it is often useful to explore alternate scenarios. For example, in

an automotive application, one might want to activate a collection of rules that tests for low voltage and another set that tests for absence of fuel. Each ruleset will need to run on an individual **version** of the same underlying relation. Although the current POSTGRES version system supports alternate versions, it requires each version to have a unique name. This makes scenario construction difficult, because each scenario has to have different specifications of common rulesets. Hence, it appears that the POSTGRES version system should be expanded to include the following notions:

1) Allow versions to be defined with the same name as the relations they are based on, appended with revision numbers. This allows application queries and rules to be used on versions without the need for rewrite to accommodate different relation names.

2) Provide a way to declare the default (current) version of a relation and an (overriding) alternate current version of a relation.

3) Allow the definition of a version to automatically include several relations at once, freeing the user from needing to keep track of which relations are involved in a logical set of relations.

4) Allow rules to refer to versions.

Rulesets can be used in conjunction with such a versioning facility to produce rule customization for dataset experimentation. When an application creates a new version of a relation set, the new version inherits all of the rules of the old version. Then the application can delete unwanted rules from and add new rules to the new version, and deactivate the rulesets that it is not interested in.

6. SUMMARY

The modifications we have suggested -- an improved syntax, the introduction of rulesets, and flexible versioning capabilities -- are meant to improve the usability of the POSTGRES rules system. These change the existing design only in minor ways, and we expect to have PRS II running quickly.

REFERENCES

[BORG85] Borgida, A., "Language Features for Flexible Handling of Exceptions in Information Systems," ACM-TODS, Dec. 1985.

[DATE81] Date, C., "Referential Integrity," Proc. Seventh International VLDB Conference, Cannes, France, Sept. 1981.

[DAYA88] Dayal, U. et. al., "The HiPAC Project: Combining Active Databases and Timing Constraints," SIGMOD Record, Vol. 17, No. 1, March 1988.

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

[HELD75] Held, G. et. al., "INGRES: A Relational Data Base System," Proc 1975 National Computer Conference, Anaheim, Ca., June 1975.

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

[STON88] Stonebraker, M. et. al., "The POSTGRES Rules System," IEEE Transactions on Software Engineering, Dec. 1988.

[STON86a] Stonebraker, M. and Rowe, L., "The Design of POSTGRES," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

[STON86b] Stonebraker, M. et. al., "An Analysis of Rule Indexing Implementations in Data Base Systems," Proc. 1st International Conference on Expert Data Base Systems, Charleston, S.C., April 1986.

[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.

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

[MCCUN86] McCune, B., et. al., “RUBRIC: A System for Rule-Based Information Retrieval,” IEEE Transactions on Software Engineering, Vol. SE-11, No. 9, Sept. 1985.

[WENS88] Wensel, S. (ed.), “The POSTGRES Reference Manual,” Electronics Research Laboratory, University of California, Berkeley, CA, Report M88/20, March 1988.