THE POSTGRES RULE MANAGER

Michael Stonebraker, Eric Hanson and Spyros Potamianos

EECS Department
University of California
Berkeley, Ca., 94720

Abstract

This paper explains the rule subsystem that is being implemented in the POSTGRES DBMS. It is novel in several ways. First, it gives to users the capability of defining rules as well as data to a DBMS. Moreover, depending on the scope of each rule defined, optimization is handled differently. This leads to good performance both in the case that there are many rules each of small scope and a few rules each of large scope. In addition, rules provide either a forward chaining control flow or a backward chaining one, and the system will choose the control mechanism that optimizes performance in the cases where it is possible. Furthermore, priority rules can be defined, thereby allowing a user to specify rule systems that have conflicts. This use of exceptions seems necessary in many applications. Lastly, our rule system can provide database services such as views, protection, integrity constraints, and referential integrity simply by applying the rules system in particular ways. Consequently, no special purpose code need be included in POSTGRES to handle these tasks.

1. INTRODUCTION

There has been considerable interest in integrating data base managers and software systems for constructing expert systems (e.g. KEE [INTE85], Prolog [CLOC81], and OPS5 [FORG81]). Although it is possible to provide interfaces between such rule processing systems and data base systems (e.g. [ABAR86, CERI86]), such interfaces will only perform well if the rule system can easily identify a small subset of the data to load into the working memory of the rule manager. Such problems have been called ``partitionable´´ Our interest is in a broad class of expert systems which are not partitionable.

An example of such a system would be an automated system for trading stocks on some securities exchange. The trading program would want to be alerted if a variety of

This research was sponsored by the National Science Foundation under Grant DMC-8504633 and by the Navy Electronics Systems Command under contract N00039-84-C-0039.

data base conditions were true, e.g. any stock was trading excessively frequently, any stock or group of stocks was going up or down excessively rapidly, etc. It is evident that the trading program does not have any locality of reference in a large data base, and there is no subset of the data base that can be extracted. Moreover, even if one could be identified, it would be out of date very quickly. For such problems, rule processing and data processing must be more closely integrated.

There are many mechanisms through which this integration can take place. In this paper we indicate a rather complete rules system which is quite naturally embedded in a general purpose data base manager. This next-generation system, POSTGRES, is described elsewhere [STON86a]; hence we restrict our attention in this paper solely to the rules component.

There are three design criteria which we strive to satisfy. First, we propose a rule system in which conflicts (or exceptions [BORG85]) are possible. The classic example is the rule ``all birds fly´´ along with the conflicting exception ``penguins are birds which do not fly´´ Another example of conflicting rules is the situation where all executives have a wood desk. However, Jones is an executive who uses a steel desk. It is our opinion that a rule system that cannot support exceptions is of limited utility.

The second goal of a rule system is to optimize processing of rules in two very different situations. First, there are applications where a large number of rules are potentially applicable at any one time, and the key performance issue is the time required to identify which rule or rules to apply. The automated stock trader is an example application of a rule system with a large number of rules each of narrow scope. Here, the system must be able to identify quickly which (of perhaps many) rules apply at a particular point in time. On the other hand, there are applications where the amount of optimization used in the processing of exceptionally complex rules is the key performance indicator. For example, consider the rule which derives the ANCESTOR relation from a base relation

PARENT (person, offspring)

If the user asks a query of the ANCESTOR relation, then rule processing will generate a recursive query to the PARENT relation. The key task is to optimize this resulting query and research in this area is presented in [BANC86]. A general purpose rules system must be able to perform well in both kinds of situations.

The third goal of a rules system embedded in a data manager should be to support as many data base services as possible. Candidates services include integrity control, referential integrity, transition constraints, and protection. As noted in [STON82], the code needed to perform these tasks correspond to small special purpose rules systems. A robust rule system should be usable for these internal purposes, and the POSTGRES system achieves this goal.

In Section 2 of this paper we discuss the syntax of POSTGRES rules and the semantics desired from a rule processing engine. Then, in Section 3 we discuss two optimization issues. First, the time at which a rule can be awakened can be varied, and provides a valuable opportunity for performance improvement. Secondarily, the mechanism that is used to ``fire´´ rules can be used at multiple granularities and will be a second optimization possibility. In Section 4 we explain the different kinds of locks which POSTGRES must set to support rule processing. Sections 5 and 6 then turn to the algorithms which the rule manager runs when the various locks are encountered during command processing. Lastly, Section 7 indicates how our rules system can be used to support views, protection, and integrity control.

2. POSTGRES RULE SEMANTICS

2.1. Syntax of Rules

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 [STON86a, 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 noted.

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 is 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. One implementation will be to wake up the above command whenever Bill's salary changes so the salary alteration can be propagated to Mike. This implementation resembles previous proposals [ESWA76, BUNE79] to support **triggers,** and efficient wake-up services are a challenge to the POSTGRES implementation. A second implementation will be to delay evaluating the rule until a user requests the salary of Mike. With this implementation, rules appear to utilize a form of lazy evaluation [BUNE82].

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´´
```

The second tag which can be applied to any POSTQUEL command is ``refuse´´. For example, the above retrieve command can be turned into this second kind of rule as follows:

```
refuse retrieve (EMP.salary) where EMP.name = ``Mike´´
```

The semantics of a refuse command is that it should NEVER be run. Hence, if any subsequent request for Mike's salary occurs, POSTGRES should refuse to access it. More precisely, the semantics of any command with a refuse modifier is that the indicated operation cannot be done to any tuple which satisfies the qualification. For qualifications spanning more than one relation, the qualification is true if values for the tuple in question are substituted into the qualification and the result evaluates to true. Syntactically, append and delete commands do not contain a target list when tagged with ``refuse´´, while replace and retrieve commands contain only a list of attributes.

Rules with a refuse modifier are generally useful for protection purposes; for example the following rule denies Bill access to Mike's salary.

```
refuse retrieve (EMP.salary)
where EMP.name = ``Mike´´
and user() = ``Bill´´
```

In this command, user() is a POSTGRES function which returns the login name of the user who is running the current query. Commands with a refuse modifier are also useful for integrity control. For example, the following rule refuses to insert employees who earn more than 30000.

```
refuse append to EMP where EMP.salary > 30000
```

One final example illustrates integrity control using a refuse modifier. The following rule disallows the deletion of a department as long as there is at least one employee working in the department. This corresponds to one situation that arises in referential integrity [DATE81].

```
refuse delete DEPT where DEPT.dname = EMP.dept
```

The final tag which can be applied to a POSTQUEL command is the modifier ``one-time´´. For example:

```
one-time replace EMP (salary = E.salary) using E in EMP
where EMP.name = ``Mike´´ and E.name = ``Bill´´
```

The semantics of this command is that it should be done exactly once when the qualification is true. In this case, the effect is exactly the same as if the command was submitted directly by a user with no modifier. However, the following example shows the utility of this kind of rule in providing so-called ``one shots´´.

```
one-time retrieve (EMP.salary)
where EMP.name = ``Mike´´
and time() > ``April 15´´
```

This command will be run once after April 15th to retrieve Mike's salary.

There is great leverage in these three simple rule constructs. However the semantics of always and one-time commands present a problem as explored in the next subsection.

2.2. Semantics of Always and One-time Rules

Always and one-time rules share a common semantic problem which can be illustrated by the following rules that provide a salary for Mike.

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

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

The first rule assigns to Mike the salary of Fred while the second rule assigns to Mike the salary of Bill. There are several possible outcomes which might be desired from this conflicting collection of rules. The first option would be to reject this set of rules because it constitutes an attempt to assign two different values to the salary of Mike. Moreover, these two commands could be combined into a single POSTQUEL update, e.g.:

```
always replace EMP (salary = E.salary)
where EMP.name = ``Mike´´
and (E.name = ``Bill´´ or E.name = ``Fred´´)
```

Such updates are **non-functional** and are disallowed by most data base systems (e.g INGRES [RTI85]) which detect them at run time and abort command processing. Hence the first semantics for always and onetime rules would be to demand functionality and refuse to process non-functional collections.

Of course functionality is not always desirable for a collection of rules. Moreover, as noted in [KUNG84], there are cases where non-functional updates should be allowed in normal query processing. Hence, we now turn to other possible definitions for this rule collection.

The second definition would be to support **random** semantics. If both rules were run repeatedly, the salary of Mike would cycle between the salary of Bill and that of Fred. Whenever, it was set to one value the other rule would be run to change it back. Hence, a retrieve command would see one salary or the other depending on which rule had run most recently. With random semantics, the user should see one salary or the other, and POSTGRES should ensure that no computation time is wasted in looping between the values.

The third possibility would be to support **union** semantics for a collection of rules. Since POSTQUEL supports columns of a relation of data type procedure, one could define salary as a procedural field. Hence, commands in POSTQUEL would be the value of this field and would generate the ultimate field value when executed. In the salary field for Mike, the following two commands would appear:

```
retrieve (EMP.salary) where EMP.name = ``Bill´´
retrieve (EMP.salary) where EMP.name = ``Fred´´
```

If Mike's salary was retrieved, both Fred's salary and Bill's salary would be returned. Hence, when multiple rules can produce values, a user should see the union of what the rules produce if union semantics are used.

To support exceptions, one requires a final definition of the semantics of rules, namely **priority** semantics. In this situation, a priority order among the rules would be established by tagging each one with a priority. Priorities are unsigned integers in the range 0 to 15, and may optionally appear at the end of a command, e.g:

```
always replace EMP (salary = 1000) where EMP.name = ``Mike´´ at priority = 7
```

If a priority is not specified by a user, then POSTGRES assumes a default of 0. When more than one rule can produce a value, POSTGRES should use the rule with highest priority. For example, suppose the priority for the ``Fred´´ rule is 7 and for the ``Bill´´ rule is 5. Using priority semantics the salary of Mike should be equal to the salary of Fred.

Since one of the goals of the POSTGRES rules systems is to support exceptions, we choose to implement priority semantics. Hence a user can optionally specify the relative priorities of any collection of tagged commands that he introduced, and the highest priority rule will be used in cases where a conflict exists. If multiple rules have the same priority then POSTGRES chooses to implement random semantics for conflicting rules, and the result specified by any one of them can be returned. It would have been possible (in fact easy) to insist on functional semantics. However, we feel that this is a less useful choice for rule driven applications.

Notice that collections of rules can be defined which produce a result which depends on the order of execution of the rules. For example, consider the following rules:

```
always delete EMP where EMP.salary = 1000

always replace EMP (salary = 2000)
where EMP.name = ``Mike´´
```

If Mike receives a salary adjustment from 2000 to 1000, then the delete would remove him while the replace would change his salary back to 2000. The final outcome is clearly order sensitive. If these commands were run concurrently from an application program, then two outcomes are possible depending on which command happened to execute first. In an analogous fashion, rules are awakened in a POSTGRES determined order, and the ultimate result may depend on the order of execution.

It is also possible for a user to define ill-formed rule systems, e.g.:

```
always replace EMP (salary = 1.1 * E.salary) using E in EMP
where EMP.name = ``Mike´´
and E.name = ``Fred´´

always replace EMP (salary = 1.1 * E.salary) using E in EMP
where EMP.name = ``Fred´´
and E.name = ``Mike´´
```

This set of rules says Fred makes 10 percent more than Mike who in turn makes 10 percent more than Fred. Clearly, these rules will never produce a salary for either Mike or Fred. In such situations, the goal of POSTGRES is to avoid going into an infinite loop.

The algorithms we use are discussed in Sections 5 and 6.

We now turn to a discussion of the optimization tactics which POSTGRES employs.

3. OPTIMIZATION OF RULES

3.1. Time of Awakening of Always and One-time Commands

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´´
```

Clearly Mike's salary must be set to Bill's which must be 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 POSTGRES is to perform early evaluation of rules, and a forward chaining control flow will result.

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 produce it requiring in turn the second rule to be run to obtain needed data. This control structure is known as **backward chaining,** and we will term it **late** evaluation. The second option available to POSTGRES is to delay evaluation of a rule until a user requires something it will write. At this point POSTGRES must produce the needed answer as efficiently as possible using a backward chaining control flow.

Clearly, the choice of early or late evaluation has important performance consequences. If Fred's salary is updated often and Mike's and Bill's salaries are read infrequently, then late evaluation is appropriate. If Fred does not get frequent raises, then early evaluation may perform better. Moreover, response time to a request to read Mike's salary will be very fast if early evaluation is selected, while late evaluation will generate a considerably longer delay in producing the desired data. Hence, response time to user commands will be faster with early evaluation. The choice of early or late evaluation is an optimization which POSTGRES will make internally in all possible situations, and we discuss our approach to this important problem in Section 7.

There are two important restrictions which limit the choice that POSTGRES can make. First, fields which are written by late rules cannot be indexed, because there is no way of knowing what values to index. For example, a secondary index on the salary column of EMP cannot be constructed if there are any late rules which write salary data. On the other hand, early rules can write fields which are indexed.

A second restriction concerns the mixing of late and early rules. Consider, for example, the situation where the Bill-to-Mike salary rule is evaluated early while the Fred-to-Bill salary rule is evaluated late. A problem arises when Fred receives a salary adjustment. The rule to propagate this adjustment on to Bill will not be awakened until somebody proposes to read Bill's salary. On the other hand, a request for Mike's salary will retrieve the old value because there is no way for the Bill-to-Mike rule to know that the value of Bill's salary will be changed by a late rule. To avoid this problem, POSTGRES must ensure that no late rule writes a data item read by an early rule.

To deal with these two restrictions, POSTGRES takes the following precautions. Every column of a POSTGRES relation must be tagged as ``indexable´´ or ``non-indexable´´. Indexable columns cannot be written by late rules, while non-indexable columns permit late writes. To ensure that no late rule writes data read by an early rule, POSTGRES enforces the restriction that early rules cannot read data from non-indexable columns. The POSTGRES implementation has the parser produce two lists of columns for each rule, those in the target list to the left of an equals sign and those appearing elsewhere in the rule. These lists are respectively the write-set and read-set for the rule. If the read-set contains an indexable field, we tag the rule ``read I´´. Similarly, a rule that writes an indexed field is tagged ``write I´´. For non-indexed fields, the corresponding tags are ``read NI´´ and ``write NI´´. Table 1 shows the allowable execution times for the various rule tags.

The consequences of Table 1 are that some rules are not allowable, some must be evaluated early, some must be evaluated late, and some can be evaluated at either time. This last collection can be optimized by POSTGRES. In a well designed data base we expect many rules to read indexed fields for fast access. Hence, if they write non-

		read tag		
		I	NI	I,NI
	I	early	not permitted	not permitted
write	NI	early/late	late	late
tag	I,NI	early	not permitted	not permitted

Time of Rule Awakening

Table 1

indexable fields they are optimizable. On the other hand, all inserts and deletes must be early because relations will tend to have at least one indexable field.

Within these constraints and considerations, POSTGRES will attempt to optimize the early versus late decision on a rule by rule basis. Not only will a decision be made when a rule is first inserted, but also an asynchronous demon, REVEILLE/TAPS (Rule EValuation EIther earLy or LatE for the Trigger Application Performance System), will run in background to make decisions on which rules should be converted from late to early execution and vice-versa. The architecture of REVEILLE/TAPS is currently under investigation.

3.2. Granularity of Locking for Rules

POSTGRES must wake-up rules at appropriate times and perform specific processing with them. In [STON86b] we analyzed the performance of a rule indexing structure and various structures based on physical marking (locking) of objects. 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 ascertains all data items that are read or proposed for writing. On each such data item or its enclosing column, it sets one of 13 kinds of locks which are detailed in the next section. Later when a user reads or writes one of these marked objects, appropriate processing can be triggered as explained in Sections 5 and 6.

Locks are typically set at the attribute level. However, there are situations where lock escalation may be desirable. For example, consider the rule:

```
always replace EMP (salary = avg (EMP.salary where EMP.dept = ``shoe´´))
where EMP.name = ``Mike´´
```

This rule will read the salaries of all shoe department employees to compute the aggregate. Rather than setting a large number of attribute locks, it may be preferable to **escalate** to a column level lock.

POSTGRES will choose either fine (attribute) granularity or coarse (column) granularity as an optimization issue. It can either escalate after it sets too many fine granularity locks or guess at the beginning of processing based on heuristics. The current wisdom for conventional locks is to escalate after a certain number of locks have been set [GRAY78, KOOI82]. For simplicity in the first implementation, POSTGRES will guess the granularity in advance. Dynamic escalation is left as a future enhancement.

Rule locks differ from normal read and write locks in several ways. First, normal locks are set and released at high frequency and exist in relatively small numbers. When a crash occurs, the lock table is not needed because recovery can be accomplished solely from the log. Hence, virtually all systems utilize a main memory lock table for normal locks. On the other hand, locks set by rules exist in perhaps vast numbers since

POSTGRES must be prepared to accommodate a large collection of rules. Secondly, locks are set and reset at fairly low frequency. They are only modified when rules are inserted, deleted, their time of evaluation is changed, or in certain other cases to be explained. Lastly, if a crash occurs one must not lose the locks set by rules. The consequences of losing rule locks is the requirement that they be reinstalled in the data base and recovery time will become unacceptably long. As a result, rule locks must persist over crashes.

Because of these differences, we are storing rule locks as normal data in POSTGRES tuples. This placement has a variety of advantages and a few disadvantages. First, they are automatically persistent and recoverable and space management for a perhaps large number of locks is easily dealt with. Second, since they are stored data, POSTGRES queries can be run to retrieve their values. Hence, queries can be run of the form ``If I update Mike's salary, what rules will be affected?´´ This is valuable in providing a debugging and query environment for expert system construction. The disadvantage of storing the locks on the data records is that setting or resetting a lock requires writing the data page. Hence, locks associated with rules are expensive to set and reset.

The decision on lock granularity in this new context has a crucial performance implication. In particular, one does not know what attribute level locks will be observed during the processing of a query plan until specific tuples are inspected. Hence, if late evaluation is used, one or more additional queries may be run to produce values needed by the user query. Consequently, in addition to the user's plan, N extra plans must be run which correspond to the collection of N late rules that are encountered. These N+1 queries are all optimized separately when attribute level locks are used. Moreover, these plans may awaken other plans which are also independently optimized.

On the other hand, if all locks are escalated to the column level, the query optimizer knows what late rules will be utilized and can generate a composite optimized plan for the command as discussed in Section 6. This composite plan is very similar to what is produced by query modification [STON75] and is a simplified version of the sort of processing in [ULLM85]. It will sometimes result in a more efficient total execution. However, setting column level locks has an important performance disadvantage. For example, if the rule noted earlier that sets Mike's salary to that of Bill uses early execution and sets column level locks, , then **ALL** incoming update commands will awaken the rule whether or not they write Bill's salary. This will result in considerable wasted overhead in trying to use rules which don't apply.

Like the decision of early versus late evaluation, the decision of lock granularity is a complex optimization problem. Initial investigation [HONG87] suggests that record level locking is preferred in a large variety of cases; however a more detailed study is underway.

Unfortunately, there appears to be no way to prioritize two commands which lock at different granularities. Hence, priorities can only be established for collections of column locking rules or attribute locking rules.

4. SETTING LOCKS

4.1. Introduction

POSTGRES rules are supported by setting various kinds of locks as noted in the previous section. One-time rules are the same as always rules except that there is an automatic deletion of the rule when a successful firing takes place. The only special case code required for one-time commands pertains to ones with a time clause. For those, POSTGRES will perform an insert into a calendar relation and have a system demon which will wake up periodically and see if there are rules in the calendar to awaken. Consequently, we will concentrate on always and refuse rules.

When an early rule is installed, it must set early-read and early-write locks on all objects that it reads and writes respectively. Moreover, late rules must set similar late-read and late-write locks. However, it will be desirable to distinguish three different kinds of read locks for the following three situations.

Consider the rule which propagates Fred's salary on to Bill, i.e:

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

If this rule is evaluated early and Fred's salary changes, then this rule must be awakened to propagate the change on to Bill. Clearly, no new objects will be read or written because of this salary adjustment. Hence, the recalculation of Bill's salary is the only task which must be accomplished, and no locks will change. Fred's salary field will be marked with an R1 lock to indicate this cheapest mode of rule wake-up.

On the other hand, suppose that Bill does not exist as an employee yet. Obviously, this rule will not be able to give Bill a salary. However, at the time he is inserted, the rule must be awakened to give him a salary. In this case, the rule must be run but the only locks affected will be on the tuple just inserted. This second wake-up mode is indicated by placing an R2 lock on the name of Bill. Lastly, if Fred is not yet an employee, then clearly the rule cannot propagate a salary on to Bill. When Fred is inserted, the rule must wake up to do the appropriate salary modification and must also set locks on records in the data base other than the one just updated. This third wake-up mode is indicated by placing an R3 lock on the name of Fred.

As a result, always commands can set the following locks:

ER1:	early read lock -- cheapest wake-up
ER2:	early read lock -- more expensive wake-up
ER3:	early read lock -- most expensive wake-up
EW :	early write lock
LR1:	late read lock -- cheapest wake-up
LR2:	late read lock -- more expensive wake-up
LR3:	late read lock -- most expensive wake-up
LW :	late write lock

Refuse rules will set late-read locks in the same way as always commands. However, they must also set special kinds of write locks on objects they propose to change. These locks are:

RR: refuse retrieve
RA: refuse append
RD: refuse delete
RU: refuse update
RE: refuse execute

The next three subsections discuss how these 13 kinds of locks get set and reset.

4.2. Set-up Needed

When a refuse or always command is entered by a user, the query tree corresponding to the new rule must be decorated with a read marker or a write marker on certain nodes. For each node which corresponds to an attribute in some relation, the parser must place markers as follows:

read markers:

R1: attributes on right hand side of an assignment in the target list
R2: any attribute in the qualification with the same
tuple variable as the relation being updated
R3: other attributes in qualification

write markers:

W : all attributes on left hand side of a target list assignment for always commands
RA: the relation affected for refuse append command
RD: the relation affected for refuse delete commands
RE: all attributes in the target list for refuse execute commands
RR: all attributes in the target list for refuse retrieve commands
RU: all attributes in the target list for refuse replace commands

If a field name appears more than once in the qualification then each marker must identify the particular node in the tree that it is associated with.

Lastly, the parser must tag the rule with ``early´´ ``late´´ ``either´´ or return an error message according to Table 1 of the previous section.

4.3. Insertion of Rules

REVEILLE/TAPS will make the early/late decision for always commands with a status of ``either´´. and the lock granularity decision for all rules. If a complete scan of any relation is done, column level locking will be used. Otherwise, REVEILLE/TAPS can freely choose the granularity. Next, POSTGRES will insert an entry into a rule relation in the system catalogs containing the code for the rule, its execution time, its lock granularity and assorted other information. POSTGRES will now change the decorations on the parse tree to EW, ER1, ER2, and ER3 for early rules and LW, LR1, LR2, and LR3 for late rules. The command will then be optimized and executed normally. During each scan of a relation, the attributes being accessed are identified in the plan. Hence, a ‘‘lock structure’’ can be built containing each locked attribute along with its type of lock, the rule identifier and the rule priority.

If relation granularity has been chosen, this lock structure will be placed in the RELATION relation tuple for the relation being accessed. If early evaluation has been chosen, a modified command will be run to update appropriate data values. This command is constructed by negating the qualification of all higher priority rules and ANDing them onto the rule qualification. For late rules, no such extra processing is done.

If record level granularity has been selected, the processing is somewhat more complex because of the necessity of correctly deal with so-called ‘‘phantoms’’. For example, consider an early rule to set Mike’s salary to be the same as Bill’s. If Bill is not yet an employee, then the rule has no effect. However, when Bill is hired, the rule must somehow be awakened to propagate his salary. As a result a means must be found for the inserted record to inherit the appropriate rule lock, so that appropriate rule processing can be triggered.

POSTGRES solves this problem by placing the lock structure both on accessed data records and on accessed index records. In addition, POSTGRES inserts a ``stub record´´ in the index to denote the beginning and end of a scan, which it also tags with the lock structure.

In addition, inheritance of locks must take place during normal command processing. For example, the above Bill-to-Mike rule rule would insert a stub record in the B-tree index for names, assuming for the moment that Bill is not yet an employee. At some later time a POSTGRES command will insert Bill’s record along with his corresponding index record. POSTGRES must ensure that both the data record and the index record **inherit** the locks from all index records adjacent to the inserted index record, in this case from the stub record.

The above mechanism must be adjusted slightly to work correctly with hashed secondary indexes. In particular, a secondary index record must inherit all locks in the same hash bucket. Hence, ``adjacent´´ must be interpreted to mean ``in the same hash bucket´´.

This mechanism is essentially the same one used by System R to detect phantoms. Although cumbersome and somewhat complex, it appears to work and no other alternative is readily available. Since POSTGRES supports user-defined secondary indexes [STON86c], this complexity must be dealt with by index code written by others.

Lastly, if the rule is an always command with early execution, POSTGRES must calculate the proposed data values and place them in all data records for which there is no higher priority EW lock.

4.4. Deletion of a Rule

To delete a rule, the run-time system must execute the rule in a special mode to find all the lock packages set on behalf of the rule. Then, it must update all such data and index records to remove the locks. Finally, other rules with EW locks on fields written by the deleted rule must be awakened.

5. RECORD LEVEL LOCK PROCESSING

The execution routines in POSTGRES must perform certain actions when a tuple tagged with a lock structure is retrieved, modified, deleted, inserted or executed during the normal processing of a user command. Certain actions are performed by the run-time system while the remainder are executed by a special module called the ``rule manager´´. We discuss each module in turn.

5.1. Processing in the Run-time System

When a tuple is inserted, the appropriate keys must be inserted into all secondary indexes. These secondary index records plus the data record must inherit all appropriate lock structures as noted in the previous section. Now the tuple with all its proposed lock structures must be passed to the rule manager.

When a tuple is to be deleted, the tuple together with all its locks will be passed to the rule manager for processing. When a collection of fields in a tuple are retrieved or executed, the appropriate fields and their lock structures must be passed to the rule manager.

When a tuple is modified, all the changes must be installed in the appropriate secondary indexes and new locks must be inherited as in the case of insertions. In addition, all lock structures that were deleted by the index deletions must be noted. A data structure will be passed to the rule manager consisting of:

the old values of the updated fields
the locks to be deleted from the updated fields
the new values of the updated fields
the continuing locks on the updated fields
the locks to be added to the updated fields
the fields which are not being updated

The rule manager must perform the actions indicated in the following subsection.

5.2. The Rule Manager

The rule manager processes inserted, deleted, retrieved, executed, and replaced tuples and returns a revised tuple or an error message to the execution routine. For inserts and deletes, it looks at all fields. For each one with a lock, it does the action indicated in Tables 2 and 3 below. For retrieves and executes, it looks only at the fields retrieved or executed, and does the action indicated in the tables below. For replaces, things are a bit more complex. The rule manager must process the refuse replace locks first according to Table 2. Then, it should process all the continuing locks on the updated fields according to the replace column in Table 3. The last step is to process the new locks and the no longer valid locks using the append and delete columns respectively in Table 3.

In Table 3 there are no actions to take when LR1 or LR2 locks are observed; hence their is no row for them and they need never be set. In Tables 2 and 3, the symbols have the following meaning:

Refuse-Lock	retrieve	execute	replace	delete	append
RR	a				
RE		a			
RU			a		
RD				a	
RA					a

Actions for Refuse Locks

Table 2

Always-Lock	retrieve	execute	delete	append	replace
EW				b	c
LW	d	d			
ER1			e	e	f
ER2					g
ER3 or LR3			h	i	j

Actions for Always Locks

Table 3

a) Generate an error message for the executor if the tuple satisfies the qualification.

b) Check if the tuple actually satisfies the rule. If not remove the EW lock. Take the value returned by the highest priority rule and put it in the tuple. If the highest priority rule is a delete, then remove the tuple.

c) Refuse the offered value unless it is made on behalf of the rule holding the lock or a higher priority rule.

d) substitute the current tuple into the query plan for the rule and run the rule as a retrieve command. Take the first returned value and plug it into the tuple as a value, thereby implementing random semantics.

For example, consider a query to retrieve the salary of Bill and a late rule that ensures Bill's salary is the same as that of Fred, i.e.:

always replace EMP (salary = E.salary) using E in EMP

```
where EMP.name = ``Bill´´ and E.name = ``Fred´´
```

In this case the user read of the salary field will conflict with the LW lock from the rule. The rule will be turned into the following retrieve command:

```
retrieve (salary = E.salary)
where ``Bill´´ = ``Bill´´ and E.name = ``Fred´´
```

The salary of the first Fred to be returned is placed in the record returned by the rule manager.

e) All records that have an ER1 lock must have an ER3 lock elsewhere in the tuple. In the case that a delete or insert occurs, the field having an ER3 lock will also be deleted or inserted and the processing appropriate to that stronger lock will have precedence.

f) Substitute the proposed tuple into the rule and run it as a normal command to update appropriate data items.

g) Substitute the new value of the tuple into the rule and see if the rule evaluates to true. If not remove the EW locks for the fields in this tuple associated with the ER2 lock. Execute step b: to find a replacement value for the field.

h) In this case some locks may have to be deleted. Hence, substitute the values for the current tuple into the rule, add on the qualification

```
and object-identifier = ``this-tuple´´
```

and execute it in ``rule deletion´´ mode to find the locks to delete. The second step is to reinsert locks on data items that can be found from duplicates of the deleted data item. To perform this function, the rule should be run in ``rule insertion´´ mode with the the following qualification appended:

```
and object-identifier not equal ``this tuple´´
```

For example, consider the Fred-to-Bill salary rule above and suppose that Fred is deleted. The first step is to run the following command in rule deletion mode:

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

The second step is to run the following command in rule insertion mode.

```
always replace EMP (salary = E.salary) using E in EMP
where EMP.name = ``Bill´´
and E.name = ``Fred´´ and E.OID != ``Fred‘s OID´´
```

i) In this case some locks may have to be inserted. Hence, substitute the new tuple into the rule and execute it in ``rule insertion´´ mode. Place locks and data values in records as appropriate.

j) Do both h) and i).

The transformations in i) and j) can be performed in parallel with processing the remainder of the query as long as the rule runs with an effective command identifier which is the same as the current command. This will ensure that the command does not see any of the modifications performed by rule processing. The details of why the POSTGRES storage system supports this parallelism are contained in [STON87a]. Alternatively, these modifications can be executed at the conclusion of a user command by saving them in virtual memory or in a file. If the user command writes data on a substantial number of fields holding ER3 or LR3 locks belonging to a single rule, then it may be advantageous to simply delete and reinstall the complete rule.

If both read and write locks are held on a single field by different rules, then care must be exercised concerning the order of execution. The rule manager must construct a dependency graph to control the processing order. In this graph an arc is placed from any rule holding a LW lock on a field to all the rules holding LR1, LR2 or LR3 locks. If this graph is a tree, then the rules are processed from root to leaf. If the graph is not a tree, then the rules involved in the loop are probably not well formed, and an error message will be signaled.

6. PROCESSING RELATION LEVEL LOCKS

When POSTGRES begins to process a user command which involves a relation R, it must process all the locks held at the column level on R. To do so, it checks whether the proposed command is reading or writing any column on which a rule holds a lock and uses Tables 4 and 5 to resolve the conflict. In these tables the symbols denote the following actions:

Refuse-Lock	retrieve	execute	replace	delete	append
RR	k				
RE		k			
RU			k		
RD				k	
RA					k

Table Level Refuse Locks

Table 4

Always-Lock	retrieve	execute	delete	append	replace
EW			l	l	l
LW	m	m			
ER1			n	n	n
ER2			n	n	n
ER3			n	n	n

Table Level Always Locks

Table 5

k) Add the negation of the rule qualification to the query qualification and continue.

l) The action to take is a little different depending on whether the rule holding the EW lock is an append, replace or delete command. If it is an append, then do nothing. If it is a delete, then AND the negation of the delete qualification to the user's command. If it is a replace, then two commands must be run. The first one results from ANDing the rule qualification onto the command and replacing appropriate fields in the user's target list with target list entries from the rule. The second command results from ANDing the negation of the rule qualification to the user's command. When multiple EW locks occur, process the highest priority one first. Then proceed iteratively with the next highest one, applying it to the modified command for deletes and to the second command resulting from replace rules.

m) Since only replace commands can hold LW locks, the action to take here is to run two commands. The first results by ANDing the rule qualification to the user retrieval and substituting the rule target list for appropriate elements of the user's target list. The second command results from ANDing the negation of the rule qualification onto the user command.

n) Wake up the rule after the user qualification has been ANDed onto it to refresh its values.

When both read and write locks are held on a column of a relation by different rules, then care must again be exercised in choosing the order of rule evaluation. Construct a dependency graph as in the previous section and process the rules in the appropriate order. If the graph is not a tree, signal an error.

7. DATA BASE SERVICES

7.1. Views

POSTGRES supports updatable views using procedural fields as explained in [STON87b]. However, the rules system can be used to construct two other kinds of views, **partial views,** and **read-only views.** A read-only view is specified by creating a relation, say VIEW, and then defining the rule:

```
always retrieve into VIEW (any-target-list)
where any-qualification
```

This rule can be executed either early or late if all accessed fields are indexable; otherwise, the status of the rule is late. Late evaluation leads to conventional view processing by query modification, while early evaluation will cause the view to be physically materialized. In this latter case, updates to the base relation will cause the materialization to be invalidated and excessive recomputation of the whole view will be required. In the future we hope to avoid this recomputation and instead incrementally update the result of the procedure. The tactics of [BLAK86] are a step in this direction.

On the other hand, partial views are relations which have a collection of real data fields and additionally a set of fields which are expected to be supplied by rules. Such views can be specified by as large a number of rules as needed. Moreover, priorities can be used to resolve conflicts. As a result partial views can be utilized to define relations which are impossible with a conventional view mechanism. Such extended views have some of the flavor proposed in [IOAN84].

7.2. Integrity Control

Integrity control is readily achieved by using delete rules. For example the following rule enforces the constraint that all employees earn more than 3000:

```
delete always EMP where EMP.salary < 3000
```

Since this is an early rule, it will be awakened whenever a user installs an underpaid employee and the processing is similar to that of current integrity control systems [STON75]. Alternately, a refuse append rule can be utilized to generate the same effect.

Referential integrity is easily accomplished using the mechanisms we have defined. The modes that refuse insertions and deletions can be accomplished with refuse rules as noted in Section 2.1. The other modes can all be accomplished using always rules.

7.3. Protection

Protection is normally specified by refuse rules which have a user() in the qualification. The only abnormal behavior exhibited by this application of the rules system is that the system defaults to ``open access´´. Hence, unless a rule is stated to the contrary, any user can freely access and update all relations. Although a cautious approach would default to ``closed access´´, it is our experience that open access is just as reasonable.

A useful future extension would be a rule which hides data items by returning an incorrect value. For example, consider the following rule:

```
hide EMP (salary = 0)
where EMP.name = ``Mike´´
and user() = ``Sam´´
```

This rule should be evaluated just like a refuse rule except it must return the value in its qualification instead of the one in the data record. This would allow the protection system to **lie** to users, rather than simply allow or decline access to objects. Such a facility allows greatly expanded capabilities over ordinary protection systems.

8. CONCLUSIONS

This paper has presented a rules system with a considerable number of advantages. First, the rule system consists of tagged query language commands. Since a user must learn the query language anyway, there is marginal extra complexity to contend with. In addition, specifying rules as commands which run indefinitely appears to be an easy paradigm to grasp. Moreover, rules may conflict and a priority system can be used to specify conflict resolution.

Two different optimizations were proposed for the implementation. The first optimization concerns the time that rules are evaluated. If they are evaluated early, then a forward chaining control flow results, while late evaluation leads to backward chaining. Response time considerations, presence or absence of indexes, and frequency of read and write operations will be used to drive REVEILLE/TAPS which will decide on a case by case basis whether to use early or late evaluation. Study of the organization of this module is underway. In addition, the locking granularity can be either at the tuple level or at the relation level. Tuple level locking will optimize the situation where a large number of rules exist each with a small scope. Finding the one or ones that actually apply from the collection that might apply is efficiently accomplished. On the other hand, relation level locking will allow the query optimizer to construct plans for composite queries, and more efficient global plans will certainly result. Hence, we accomplish our objective of designing a rule system which can be optimized for either case. Lastly, the rule system was shown to be usable to implement integrity control, a novel protection system and to support access to two different kinds of views.

However, much work remains to be done. Optimizing the updating of locks when data items change is complex and possibly slow. Deleting and reinserting locks should be optimized better. Moreover, the implementation is complex and difficult to understand. Hence, a simpler implementation would be highly desirable. In general, a mechanism to update the result of a procedure is required rather than simply invalidating it and recomputing it. The efforts of [BLAK86] are a start in this direction, and we expect to search for algorithms appropriate to our environment. Moreover, it is a frustration that the rule system cannot be used to provide view update semantics. The general idea would be to provide a rule to specify the mapping from base relations to the view and then another rule(s) to provide the reverse mapping. Since it is well known that non-invertible view definitions generate situations where there is no unambiguous way to map backward from the view to base relations, one must require an extra semantic definition of what this inverse mapping should be. We hope to extend our rules system so it can be used to provide both directions of this mapping rather than only one way. Lastly, we are searching for a clean and efficient way to eliminate the annoying restrictions of our rule system, including the fact that priorities cannot be used with different granularity rules, and some rules are forced to a specific time of awakening.

REFERENCES

[ABAR86] Abarbanel, R. and Williams, M., ``A Relational Representation for Knowledge Bases,´´ Proc. 1st International Conference on Expert Database Systems, Charleston, S.C., April 1986.

[BANC86] Bancilhon, F. and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing," Proc. 1986 ACM-SIGMOD Conference on Management of Data, Washington, D.C., May 1986.

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

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

[BUNE79] Buneman, P. and Clemons, E., ``Efficiently Monitoring Relational Data Bases,´´ ACM-TODS, Sept. 1979.

[BUNE82] Buneman, P. et. al., ``An Implementation Technique for Database Query Languages,´´ ACM-TODS, June 1982.

[CERI86] Ceri, S. et. al., ``Interfacing Relational Databases and Prolog Efficiently,´´ Proc 1st International Conference on Expert Database Systems, Charleston, S.C., April 1986.

[CLOC81] Clocksin, W. and Mellish, C., ``Programming in Prolog,´´ Springer-Verlag, Berlin, Germany, 1981.

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

[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.

[FORG81] Forgy, C., ``The OPS5 User's Manual,´´ Carneigie Mellon Univ., Technical Report, 1981.

[GRAY78] Gray, J., ``Notes on Data Base Operating Systems,´´ IBM Research, San Jose, Ca., RJ 2254, August 1978.

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

1975.

[HONG87] Hong, C., ``An Analysis of Rule Locking Granularities,´´ Master's Report, Computer Science Division, University of California, Berkeley, Ca., 1987.

[INTE85] IntelliCorp, ``KEE Software Development System User's Manual,´´ IntelliCorp, Mountain View, Ca., 1985.

[IOAN84] Ioannidis, Y. et. al., ``Enhancing INGRES with Deductive Power,´´ Proceedings of the 1st International Workshop on Expert Data Base Systems, Kiowah SC, October 1984.

[KOOI82] Kooi, R. and Frankfurth, D., ``Query Optimization in INGRES,´´ Database Engineering, Sept. 1982.

[KUNG84] Kung, R. et. al., ``Heuristic Search in Database Systems,´´ Proc. 1st International Conference on Expert Systems, Kiowah, S.C., Oct. 1984.

[RTI85] Relational Technology, Inc., ``INGRES Reference Manual, Version 4.0´´ Alameda, Ca., November 1985.

[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.

[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.

[STON86c] Stonebraker, M., ``Inclusion of New Types in Relational Data Base Systems,´´ Proc. IEEE Data Engineering Conference, Los Angeles, Ca., Feb. 1986.

[STON87a] Stonebraker, M., ``The POSTGRES Storage System,´´ Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[STON87b] Stonebraker, M. et. al., ``Extending a Relational Data Base System with Procedures,´´ ACM-TODS, Sept. 1987.

[ULLM85] Ullman, J., ``Implementation of Logical Query Languages for Databases,´´ ACM-TODS, Sept. 1985.