Productivity Engineering in the UNIX† Environment

AD-A179 161

The Design of the POSTGRES Rules System

Technical Report
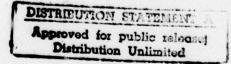
S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

87   4 8. 008

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	

4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)

6a. NAME OF PERFORMING ORGANIZATION The Regents of the University of California	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION SPAWAR
6c. ADDRESS (City, State, and ZIP Code) Berkeley, California 94720		7b. ADDRESS (City, State, and ZIP Code) Space and Naval Warfare Systems Command Washington, DC 20363-5100

8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA	8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00039-84-C-0089

8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22209	10. SOURCE OF FUNDING NUMBERS			
	PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

The Design of POSTGRES Rules System

12. PERSONAL AUTHOR(S) Michael Stonebraker, Eric Hanson and Chin-Heng Hong

13a. TYPE OF REPORT technical	13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 1986, Nov. 24	15. PAGE COUNT 19

16. SUPPLEMENTARY NOTATION

17.	COSATI CODES		18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Enclosed in paper.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR   83 APR edition may be used until exhausted.   SECURITY CLASSIFICATION OF THIS PAGE
All other editions are obsolete.

THE DESIGN OF THE POSTGRES RULES SYSTEM

Michael Stonebraker, Eric Hanson and Chin-Heng Hong

EECS Department
University of California
Berkeley, Ca., 94720

Abstract

This paper explains the rules 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 that it is possible. Furthermore, priority rules can be defined, thereby allowing a user to specify rules systems that have conflicts. This use of exceptions seems necessary in many applications. Lastly, our rule system can support an implementation of views, protection and integrity control, simply by applying the rules system in a particular way. Consequently, no special purpose code need be included 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 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 that 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. The rule whereby one derives the ANCESTOR relation from a base relation

PARENT (person, offspring)

is an example of this situation. Here, processing the rule in order to satisfy a user query to the ANCESTOR relation is the key task to optimize. 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 views, integrity control and protection. As noted in [STON82], the code needed to perform these three tasks correspond to three small special purpose rules systems. A robust rules system should be usable for these internal purposes, and the POSTGRES rules system is a step toward this goal.

In Section 2 of this paper we discuss the syntax of POSTGRES rules and the semantics desired from a rule processing engine. The general idea is to propose a mechanism that appears to the user as a trigger subsystem [ESWA76, BUNE79]. However the novel aspect of our proposal is that we have two different optimization tactics. First, the time at which triggers are evaluated can be varied. It is clear that triggers can be activated whenever something in their read set changes. However, activation can also be delayed in some cases until somebody queries a data item that they will write. Varying the time of activation will be seen to be a valuable optimization tactic. Secondly, the mechanism that is used to ``fire´´ triggers can be used at multiple granularities. This tactic will be usable to

2

optimize separately different kinds of rules. These two optimization tactics will be the subject of Section 3. Then in Section 4 we sketch the algorithms to be run when a trigger is awakened and lastly indicate in Section 5 how our rules system can be used to support protection, integrity control and view subsystems.

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, versions and time. The language is described elsewhere, and here we solely indicate the rule component of the language.

POSTQUEL supports the ability to update the salary field for the employee Mike in the EMP relation using a variation of QUEL as follows:

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

This command will set Mike's salary to that of Bill whenever it is run.

POSTQUEL also allows any insert, update, delete or retrieve command to be tagged with an ``always´´ modifier which changes its meaning. Such tagged commands become **rules** and can be used in a variety of situations as will be presently noted. For example, the following command turns the above update into a rule.

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

The semantics of this command is that it logically must 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 an **alerter.** For example, the following command will retrieve Mike's salary whenever it changes.

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

It will be shown that a variety of data base services can be provided utilizing only this ``always´´ command. These include backward chaining rules systems, forward chaining rules systems, views, protection control and integrity constraints. Hence, there is great leverage in a simple construct. However the semantics of triggers present a problem as explored in the next subsection.

2.2. Semantics of Rules

Suppose two rules have been defined that provide salaries for Mike, e.g.:

3

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

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

There are several possible outcomes which might be desired from this command collection. The first option would be to reject this collection 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.:

```
replace always 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 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 also 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 with a priority. Priorities are floating point numbers in the range 0 to 1, and may appear after the keyword **always**. The Default priority is 0. For example, suppose the priority for the ``Fred´´ 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. If priorities are not specified, then POSTGRES chooses to implement random semantics for conflicting rules, and can return the result specified by either of them. However, one of the implementations which we propose is very efficient if union semantics are utilized. Hence, we do not view returning the answers produced by both rules as an error.

In summary, if a user reads a data item for which a collection of rules can produce an answer and some subcollection of the rules have been prioritized using the PRIORITY command, then POSTGRES will return the value produced by the highest priority command from the subcollection. In addition, it may return values specified by some of the unprioritized rules. In the case that the subcollection of prioritized rules is empty, POSTGRES will return at least one value produced by one of the rules and may optionally return more than one. 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 commands:

delete always EMP where EMP.salary = 1000

replace always 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. POSTGRES does not alter these semantics in any way. Hence, rules are awakened in a POSTGRES determined order, and the ultimate result may depend on the order of execution.

3. IMPLEMENTATION OF RULES

3.1. Time of Awakening

Consider the following collection of rules:

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

replace always 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

5

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 an algorithm to be described in Section 4, and a backward chaining control flow will result.

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. However, there are three important restrictions which limit the available options.

The first restriction concerns the generality of POSTGRES procedures. A procedure written in POSTQUEL or in an arbitrary programming language can be **registered** with POSTGRES after which it can be used in the query langauge. If arrogance is such a user defined procedure, then the following is a legal rule:

retrieve always (EMP.name)
where arrogance (EMP.name) > 10

The details of this registration process are contained in [STON86c]. The only issue of concern here is that the definer must specify if the procedure is **cachable.** In the above situation, the answer returned by arrogance does not depend on the time of execution and is thereby cachable. Early or late execution is allowable for rules that contain cachable procedures. Although most procedures are cachable, there are cases where needed information is not available until run time. The following rule contains an uncachable procedure:

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

Here, ``user´´ is a procedure which makes a system call to ascertain the identity of the user who is currently executing Hence, the desired effect is that Sam should see Mike's salary as 0. Other users may see a different value. Clearly, this procedure cannot be evaluated early, since needed information is not available.

6

Such rules which deal with protection generally contain uncachable procedures and must be executed late.

A second restriction concerns indexing. Fields for which there are late rules cannot be indexed, because there is no way of knowing what values to index. Hence, 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 are compatible with indexes on fields which they update.

A third 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 rules write any data objects read by early rules.

To deal with these latter 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 reads cannot access data from non-indexable columns. To support this, the POSTGRES parser produces two lists of columns, those in the target list to the left of an equals sign and those appearing elsewhere in the rule. These lists are the write-set and read-set respectively for a 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

rule status	Time of Awakening
read I write NI	early or late
read NI write I	not permitted
read I write I	early
read NI write NI	late

Time of Rule Awakening

Table 1

7

early, some must be evaluated late, and some can be evaluated at either time. This last collection can be optimized by POSTGRES.

To achieve further optimization, POSTGRES can temporarily change the time of evaluation of any late rule to ``temporarily early´´ if the rule does not read any data written by a late rule. Similarly, an early rule can be changed to temporarily late if it does not write an object read by an early rule. If at some subsequent time these conditions become false, then the rule must revert from its temporary status back to its permanent status.

An unfortunate consequence of Table 1 is that permanent status of all inserts and deletes is early, since all relations will have at least one indexable field.

There are many examples where late evaluation and early evaluation produce different answers, as noted in the following example:

> append always to NEWEMP (name = ``Joe´´, salary = EMP.salary)
> where EMP.name = ``Mike´´

If one executes this rule early, then a new tuple will be inserted in the NEWEMP relation each time the salary of Mike changes. On the other hand, temporarily moving the evaluation to late will cause only the last salary of Mike to be available for Joe in NEWEMP. Although we feel that users typically do not wish to specify the time of rule execution, we give them the option to do so. Hence, a rule can be tagged with ``always´´ to indicate that POSTGRES can choose the time of execution, ``early´´ to specify early execution, or ``late´´ to specify late execution. This will allow the user to generate appropriate semantics in the case of commands whose outcome is sensitive to the time of rule awakening. If the user specifies ``early´´ or ``late´´ awakening, then POSTGRES will comply with the request if possible.

Within these constraints and considerations, POSTGRES will attempt to optimize the early versus late decision on a rule by rule basis. All rules will be inserted with their permanent status, and an arbitrary decision will be made for the ones whose time of awakening is optimizable. Then 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 temporarily or permanently from late to early execution and vice-versa.

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

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

> replace always 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. If the permanent status of these rules is early, then execution of both rules will generate an infinite loop. On the other hand, late execution of the rules will result in a user query to find the salary of Fred or Mike never finishing. In either case POSTGRES must try to detect the infinite loop.

8

How to do this remains to be studied.

3.2. Granularity of Locking

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.

When a rule is installed into the data base for either early or late evaluation, POSTGRES is run in a special mode and sets appropriate locks on each tuple that it reads or proposes to write in evaluating the rule. These locks are:

      ER (column) : rule read lock by an early rule
      LR (column) : rule read lock by a late rule
      EW (column) : rule write lock by an early rule
      LW (column) : rule write lock by a late rule

These 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.

Like normal locks, there is a phantom problem to contend with. For example, consider the 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 be awakened to propagate his salary. Setting locks on tuples and attributes will not accomplish the desired effect because one can only lock actual data read or written. To deal with phantoms, POSTGRES also sets rule locks on each index

record that is read during query processing and on a ``stub record´´ which it inserts in the index to denote the beginning and end of a scan. Whenever a data record is inserted into a POSTGRES relation, appropriate index records must be added to each existing secondary index. The POSTGRES run time system must note all locks held on index records which are adjacent to any inserted secondary index record. Not only must these locks be inherited by the corresponding data record, but also they must be inherited by the secondary index record itself. 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 [STON86d], this complexity must be dealt with by the index code.

Locks may be set a record level granularity as noted above. However, there are situations where lock escalation may be desirable. For example, consider the rule:

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

This rule will read the salaries of all employees to compute the aggregate. Rather than setting a large number of record level locks, it may be preferable to escalate to a relation level lock. Hence, all of the above rule locks can also be set at the relation level. In this case they are tuple level locks set on the tuple in the RELATION relation which exists for the EMP relation.

With this information we can now discuss the actions which must be taken when rules are inserted. If record level granularity is selected, a late rule must set LR locks on all objects which it reads and LW locks on all objects it proposes to write. Similarly, an early rule sets appropriate ER and EW locks. Moreover, an early rule must install values for all data items it writes unless a higher priority command holds an EW or LW lock. It will be efficient to maintain the collection of EW and LW locks in priority order. Hence, when a write lock is set, it will be put in the correct position on the ordered list of EW and LW locks.

When table level locks are utilized, ER, EW, LR and LW locks are set at the relation level on individual attributes and early writes are installed in the data base. If a higher priority command writes the same relation, then the new rule is run but the qualification of the rule is modified to AND on the negation of all higher priority rules. Then the EW lock is inserted in the correct position of an ordered list of EW and LW locks on the appropriate tuple in the ATTRIBUTE relation.

There appears to be no way to prioritize two commands which lock at different granularities. Hence, priorities can only be established for collections of table locking rules or record locking rules.

When a rule locking at either granularity is deleted, its entries in the system catalogs are found and deleted along with any locks it is holding. If a rule holds an EW or LW lock on any object for which a lower priority rule holds a EW lock,

10

then the lower priority rule must be awakened to write its result.

Whenever a user reads a data item he will be returned the stored value if there are no record-level write locks or the highest priority lock is an EW lock. Otherwise, the algorithm in the next subsection is run on the commands holding LW locks in priority order until one produces a value. If none produce a value, then the value of the data item is whatever is stored in the field (if anything). This implements the correct notion of priorities for record locking. Whenever a command writes a data item on which a record level EW lock is held, the write is ignored and the command continues. Whenever, it writes a data item on which an LW lock is held, then the write succeeds normally. Lastly, whenever it writes a data item on which an ER lock is held, then the rule that set the lock is awakened as described in the next section.

If a user writes a relation on which LW locks are held at the relation level, then no special action is taken. On the other hand, if a write is performed on a relation with one or more EW locks, then the corresponding rules must be awakened to refresh the objects they write. The algorithm we use is discussed in the next section. Similarly, when a user reads a relation on which LW locks are held, then an algorithm is performed similar to query modification [STON75] which is also discussed in the next section.

POSTGRES will choose either fine granularity or coarse 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 fixed number of locks have been set [GRAY78, KOOI82].

The decision on escalation in this new context has a crucial performance implication. In particular, one does not know what record 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 record 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 relation level, the query optimizer knows what late rules will be utilized and can generate a composite optimized plan for the command as discussed in the next section. 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 often result in a more efficient total execution.

On the other hand, if the rules noted earlier that set Mike's and Bill's salaries are escalated to the relation level, then ALL incoming commands will use the rules whether or not they read Mike's or Bill's salary. This will result in considerable wasted overhead in using rules which don't apply.

Like the decision of early versus late evaluation, the decision of lock granularity is a complex optimization problem. Detailed study of both problems in underway.

11

4. Conflict Processing

4.1. Record Level Locks

A rule is awakened whenever a user writes a data item on which an ER or LR lock is held or reads a data item on which a rule is holding an LW lock. We treat these two cases in turn.

4.1.1. User Writes

There are three different actions which may be taken on user writes. First, the rule must note whether its collection of locks will change as a result of the write. For example, consider the following rule:

>replace always EMP (salary = 100) using E in EMP
>where EMP.age = E.age
>and E.name = ``Mike´´

This rule holds a read lock on the age of Mike and a conflict will be generated if a user writes a new value for Mike's age. In this case, the collection of employees who receive a salary adjustment changes. In this case the rule must be deleted and then reinstated. A rule which has a join field which is updated by a user must receive this first treatment.

The second action to perform is the conventional case where the set of locks does not change as a result of the update. In this case, there is no action to take for a rule with late evaluation. The third action applies to rules performing early evaluation which must be awakened and run. This can simply occur at the end of a scan of a relation; however, it will usually be faster to apply query modification to the rule to restrict its scope. One simply substitutes the values written by the user command for the current tuple into the rule target list and old values for the current tuple into the rule qualification and then wakes it up.

For example, consider a salary adjustment for Fred and the rule which propagates Fred's salary on to Bill, i.e:

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

If XXX is the proposed salary for Fred, then this rule will be awakened as:

>replace EMP (salary = XXX)
>where EMP.name = ``Bill´´
>and ``Fred´´ = ``Fred´´

4.1.2. User Reads

When a user reads a data item whose highest priority lock is an LW lock, its value must be obtained from the rule. It is probably wise to call REVEILLE/TAPS to ascertain if one of the LW locks can be profitably turned into a EW lock. If so, the user command can simply continue by utilizing the value written in the record by the early rule. If caching is unprofitable or the rule is uncachable, then the following algorithm should be run.

If the rule is an APPEND which REVEILLE/TAPS changed to temporarily late, then the command is run as a retrieve to materialize one or more tuples which are processed by the run-time POSTGRES system before proceeding. If the rule is a DELETE which was similarly changed to late, then the qualification of the DELETE is tested against the current tuple. If a match is observed, then the run-time system passes over the tuple and continues. If the rule is a REPLACE, then the command is run as a retrieve to provide possibly new values for the current tuple. This modified tuple is then passed to higher level software for further processing.

Note however, that values from the current tuple must be substituted into any such command before it is run. For example consider the following rule:

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

If the user issue a query:

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

then the rule will be awakened as:

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

If a salary is returned, then it is used in place of a salary in the tuple (if any) before passing the tuple on to higher level software.

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. If multiple fields have this property, then the corresponding rules may have to run more than once and infinite loops are possible. Efficient algorithms for this situation are still under investigation.

4.2. Relation Level Locking

Actions must be taken when a user writes a column of a relation on which a rule is holding a ER, LR or EW lock and when a user reads a column of a relation on which a rule is holding an LW lock. We discuss each situation in turn.

4.2.1. User Writes

If a user command, U, writes into a column on which a rule holds a read lock, there are three cases to consider as before. The rule may have to be deleted and reinserted if its collection of locks might change or it might require no action if it is evaluated late and no locks change. The third alternative is that the rule must be awakened to refresh the values which it writes. In this case, it will be profitable to restrict the scope of the rule by substituting values from the target list of U into the rule target list and qualification and then adding U's qualification onto the rule qualification. For example consider the following rule:

    replace always OLDSAL (salary = EMP.salary)
    where OLDSAL.name = EMP.name

13

If a relation level lock is set by the always command, and the following command is run:

> replace EMP (salary = 1000)
> where EMP.name = ``George´´

then the above algorithm will awaken the rule as:

> replace OLDSAL (salary = 1000)
> where OLDSAL.name = EMP.name
> and EMP.name = ``George´´

The algorithm for the third alternative must also be run if an EW lock is held on an updated column.

4.2.2. User Reads

The query optimizer can discover any relation level write locks held on a relation at the time it does query planning. The following query modification algorithm can be run prior to query compilation and an optimized plan constructed for the composite query.

Query modification for the various POSTQUEL commands is slightly different. Hence individual commands are addressed in turn when awakened by a POSTQUEL retrieve command Q. If the awakened command is an APPEND rule which has been changed to temporarily late by REVEILLE/TAPS, then the APPEND is substituted into Q to form a new command, Q'. Both Q and Q' must be run against the data base. For example, consider

> append always EMP (NEWEMP.all) where NEWEMP.age < 40

and the query

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

The result of query modification is two queries:

> retrieve (EMP.salary) where EMP.name = ``Mike´´
>
> retrieve (NEWEMP.salary)
> where NEWEMP.name = ``Mike´´
> and NEWEMP.age < 40

If the awakened command is a DELETE rule which has been changed to temporarily late, then query modification is run on Q to produce a new command Q' which negates the qualification of the rule and adds it to Q. For example consider:

> delete always EMP where EMP.salary < 40

If the above query to find Mike's salary is run, then it will be modified to:

> retrieve (EMP.salary)
> where EMP.name = ``Mike´´ and not EMP.salary < 40

If the awakened rule is a REPLACE command, then query modification must construct two commands as follows. The first command will be found by substituting target list values from the rule into Q and appending the rule qualification. The second command is constructed from the user command by appending the negated rule qualification. For example, consider the rule:

14

```
    replace always EMP (salary = 1000)
    where EMP.dept = ``shoe´´
```

A query to find Mike's salary will result in:

```
    retrieve (1000)
    where EMP.name = ``Mike´´ and
    EMP.dept = ``shoe´´

    retrieve (EMP.salary)
    where EMP.name = ``Mike´´ and
    not EMP.dept = ``shoe´´
```

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. If multiple columns have this property, recursion and infinite loops are possible, and algorithms for this case are under investigation.

5. DATA BASE SERVICES

5.1. Views

It is possible for the POSTGRES rules system to support two kinds of views, **partial views**, and **normal views**. A normal view is specified by creating a relation, say VIEW, and then defining the rule:

```
    retrieve always 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 permanent status of the rule is late and REVEILLE/TAPS may temporarily move it to early if no other rule performs late writes on data this rule reads. 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. Hence, in the future we hope to avoid the recomputation of procedures and instead incrementally update the result of the procedure. The tactics of [BLAK86] are a step in this direction.

Unfortunately normal views cannot be updated using the rules system described so far. Although extensive attempts have been made by the authors to specify the mapping from updates on a view to updates on base relations as a collection of rules, this effort has not yet yielded a clean solution.

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 [IONN84].

Moreover, all retrieves to such relations function correctly. Updates to such relations are processed as conventional updates which install actual data values in their fields, as long as all the rules are evaluated late. Propagating such values to

15

base relations becomes the job of additional rules. Specifying all the needed rules is a bit complex, but it can be accomplished.

5.2. 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 over-paid employee and the processing is similar to that of current integrity control systems [STON75]. However POSTGRES may be able to delay evaluation if that appears more efficient. In this case bad data is insertable but it can never be retrieved.

5.3. Protection

Protection is normally specified by replace rules which have a user() in the qualification, so they are non cachable and late evaluation is appropriate. 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.

Notice that this protection system is also novel in that it is possible for the system to lie to users, rather than simply allow or decline access to objects. The example rule discussed earlier, i.e:

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

is such an example. This facility allows greatly expanded capabilities over ordinary protection systems.

6. 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 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.

16

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 retrieve access to two different kinds of views.

On the other hand, much effort remains to be done. First, the rule system is not yet powerful enough to specify easily all the options desired from referential integrity [DATE81]. It cannot support transition integrity constraints (i.e. no employee raise can be more than 10 percent). Moreover, the rule system generates situations where a rule must be deleted and reinserted. This will be an exceedingly expensive operation, and means to make this more efficient are required. 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 more general algorithms. Lastly, it is a frustration that the rule system cannot be used to provide view update semantics. The general idea is 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.

[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.

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

[IONN84]   Ionnidis, 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.

[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., ``Object Management in POSTGRES using Procedures,'' Proc. 1986 International Workshop on Object-

oriented Database Systems, Asilomar, Ca., Sept 1986. (available from IEEE)

[STON86d]   Stonebraker, M., ``Inclusion of New Types in Relational Data Base Systems,´´ Proc. IEEE Data Engineering Conference, Los Angeles, Ca., Feb. 1986.

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