RULE PROCESSING WITH QUERY REWRITE

by

Khoon-San Jeffrey Goh

Memorandum No. UCB/ERL M91/52

5 June 1991

RULE PROCESSING WITH QUERY REWRITE

by

Khoon-San Jeffrey Goh

Memorandum No. UCB/ERL M91/52

5 June 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

RULE PROCESSING WITH QUERY REWRITE

by

Khoon-San Jeffrey Goh

Memorandum No. UCB/ERL M91/52

5 June 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

This paper discusses the theory and implementation of a general production rule system using Query Rewrite in POSTGRES. Such a system is useful because data management features including Integrity Constraints, Database Procedures. Query Modification Views and Materialized Views can be implemented as a few simple rules. Query Rewrite provides a highly efficient rule system on which these features can be implemented with performance similar to or better than traditional methods.

1 Introduction

[STON75] proposes that Query Rewrite may be effectively used to process views and integrity constraints in a Data Base Management System (DBMS). Subsequently, [STON90] shows that views and integrity constraints are merely special cases of a general production rules system and proposes that Query Rewrite be used to effectively process a subset of these production rules. [STON90] further shows some examples of how rules can be enforced by means of modifying the user queries, but presents neither an algorithm nor the complete semantics of such a rule processing strategy.

This paper provides a detailed report on the actual implementation of a Query Rewrite Rule Processor called the Query Rewrite System (QRS) in the POSTGRES DBMS, and describes the algorithms and semantics of the QRS.

This paper further provides detailed implementations of the following database functions in terms of QRS rules :

1. Deferred Maintenance Views.

2. Materialized Views.

3. Procedures.

2 Design of the Query Rewrite System

2.1 Syntax

Since both the Query Rewrite System (QRS) and the Tuple Level System (TLS) are part of the Postgres Rules System II (PRS2), they naturally share the same rule definiton syntax. This syntax is largely similar to [HANS89, WIDO90] except that we allow for *retrieve-rules* to be defined whereas neither [HANS89. WIDO90] tries to do so. Retrieve-rules, briefly, are those rules that dictate what processing should take place if a user attempts to "retrieve" or otherwise reference any data which is in the scope of the rule. In this section, we briefly review the syntax of PRS2 rules which were first presented in [STON90].

Rules are of the form :

define rule rule-name **is**
on (*retrieve* | *replace* | *append* | *delete*) **to** *relname.attname*
where *event – qualification*
do [instead]
action

where the *event-qualification* and *action* clauses are normal POSTQUEL

qualifications and statements respectively except that they can contain references to *current* and *new* wherever normal tuple variables are allowed.

The semantics of *current* and *new* are as follows : When tuples are accessed, updated, inserted or deleted, there is a *current* tuple (for retrieves, replaces and deletes) and a *new* tuple (for replaces and appends). If the event and the condition specified in the *event* clause are true for the *current* tuple, then the *action* clause is executed. First, however, values from fields in the *current* tuple and/or the *new* tuple are substituted into *current.column-name* and *new.column-name*

2.2 Semantics

[STON90] defines the semantics of the Tuple Level System (TLS) rule actions, but does not clearly define the semantics of Query Rewrite System (QRS) rule actions. To clarify the semantics of QRS rule actions, let us examine the following example of a rule which make's Fred's salary be the same as Joe's salary :

on retrieve **to** *emp.salary* **where** *emp.name* = "Fred"
do instead
retrieve (*e.salary*) **from** *e in emp*
where *e.name* = "Joe"

then the definition of a "Fred" tuple would become

retrieve (*emp.name*, *emp.age*, *e.salary*) **from** *e in emp*
where *emp.name* = "Fred"
and *e.name* = "Joe"

That is, "Fred" tuples are unchanged in any way except that their salaries are defined to be that of "Joe". The *action* part of the rule defines Fred's tuple to be the join of his name and age and Joe's salary. Therefore, evaluating the above rule action will produce a result that falls into one of the following three classes :

1. **No "Joe" tuples**

   Since "Fred" tuple(s) are defined to be the join of his name, age and Joe's salary, then, by QRS semantics, there is no tuple for "Fred". This result is consistent with a relational interpretation of the action, but is contrary to the semantics of the TLS which says that there will be as many Fred tuple(s) as there would have been without the rule, but that their salaries would be null.

2. **One "Joe" tuple**

   This produces as many tuples for Fred as there would have been without

the rule, except that all their salaries are logically "replaced" by Joe's salary. Here, the QRS semantics conincide with those of the TLS.

3. **Many "Joe" tuples**

   If m is the number of employees whose name is "Joe" and n is the number of employees whose name is "Fred", then the action quite logically produces $m * n$ "Fred" tuples. Again, this is a different result from the TLS which will take the salary of the first "Joe" tuple found and propagate it to all n "Fred" tuples.

Classes (1) and (2) are merely special cases of (3) where m, the number of "Joe" tuples is a constant value (in class (1), $m = 0$, and in class (2) $m = 1$). In [STON90], it was assumed that the semantics of the Query Rewrite System (QRS) would be identical to those of the TLS. As we have shown above, QRS and TLS have different semantics for classes (1) and (3). These semantic differences are unavoidable because QRS rules have inherently relational semantics whereas the TLS rules have "random" semantics. Therefore, the user is advised to select whichever system has the semantics appropriate to his application.

2.3 Inherent Limitations

There is one limitation to the power of the Query Rewrite System (QRS): It is unable to process recursive rules of any kind. This includes rules that are truly recursive as well as rules that are pseudo recursive. *Truly Recursive* rules are those rules that will invoke themselves because one or more *real* tuples that result from rule evaluation actually fall into the scope of the rule's event qualification (also called the *rule domain*). That is, $tuples_{event} \cap tuples_{action} \cap tuples_{userquery} \neq \emptyset$. In contrast, *pseudo recursive* rules are those rules which theoretically produce a range of tuples that *may* overlap its own domain : That is, $qual_{event} \cap qual_{action} \cap qual_{userquery}$ may or may not be empty, so the rule *might* invoke itself.

For example, the following *pseudo recursive* rule declares that the manager of the toy department earns twice the average salary of employees in the toy department whose age is less than 30.

define rule toy-mgr-salary **is**
on retrieve to *emp.salary*
where *emp.dept* = *"toy"* **and** *emp.manager* = *emp.name*
do instead
retrieve ($2 * average(emp.salary)$)
where *emp.dept* = *"toy"* **and** *emp.age* < 30

If a retrieve to the manager's salary is attempted, there is absolutely no way

to determine whether or not the rule will loop by inspecting the qualifications since the rule will recurse *only* if the manager's age is less than 30.

This is because if the manager is less than thirty years of age, the manager's tuple falls into the scope of the qualification of the action clause (also called the action qualification) and will therefore be retrieved again by the rule action. This then triggers the rule again, resulting in an infinite loop. In this scenario, then, we have a *truly recursive* situation.

On the other hand, if the manager is older than thirty years of age, his tuple does *not* fall into the action qualification, and therefore the rule will not be retriggered by the action. In this case, evaluating the rule shows it to be not recursive, although we cannot deduce this by just looking at the rule in the absence of the actual data.

From the above example, it is clear that the only way to tell if there is a loop is to know what the manager's age is beforehand. Unfortunately, this information is not available to the QRS since it *never* sees the data on which the queries will operate. In general, the QRS *cannot* distinguish between truly and pseudo recursive rules. At best, the QRS can detect "loops" based

on *logical* evaluation of the event, action and user qualifications. That is, if

$$qual_{event} \cap qual_{action} \cap qual_{userquery} \neq \emptyset$$

then QRS aborts the rule evaluation and declares that a loop exists even if actual query execution might prove otherwise. However, there is one problem with this method : it occasionally detects a loop where there is none. Thus, in the above example. QRS will declare a loop even if the manager is actually more than thirty years old.

3 Implementation

In this section. we provide a description of the various aspects of implementing the Query Rewrite System (QRS) which consists of two parts :

1. A rule definition module, which takes rules that conform to the rule syntax shown in section 2.1 and stores them in a system catalog called a *rewrite-rule* catalog.

2. A rule processing module, which takes a parse tree from the parser, modifies it by the applicable rules using algorithms 3.1 and 3.2 which are described on page 11. After all applicable rules have been evaluated,

the rewrite system returns a list of zero or more parse trees which correspond to the modified queries.

The parse trees which are returned by the rewrite system are then planned and executed in the same transaction context as the user query which triggered the rule.

3.1 Rule Locks

Although all meta data associated with *rewrite-rules* are stored in the *rewrite-rule* catalog, we still need some way to indicate which queries actually require rule evaluation, and what rules are applicable to an incoming query. This information is condensed into a small data structure called a *rule lock*. This section describes how these *rule locks* are stored and how they are used in the processing of rules.

3.1.1 Placing Locks

Although [STON90] describes both relation-level and tuple-level rule locks, tuple-level rule locks are not seen until the command is in execution, and therefore cannot be used by the QRS. Thus, in our implementation, we

use relation-level rule locks. These locks are stored as an attribute in the *Relation-relation* when the rule is defined.

3.1.2 Activating Locks

Currently, in the course of parsing a query involving a relation X , the parser reads in the tuple describing X from the *Relation-relation* and stores the description in the parse tree. Since the rule lock is now part of that tuple, the only additional cost is that the parser now copies the rule lock for X in addition to the other information it already copies.

This involves negligible overhead for relations that have rule locks placed on them and no overhead at all for relations that do not have any rule locks. Thus, when the QRS receives the parse tree. all relevant locking information is already in the parse tree , and the QRS only has to decide whether or not the rules are applicable to the current parse tree and process the rules if they are.

3.2 Algorithms

If the parse tree has locks placed on it, the Query Rewrite System (QRS) modifies the query according to the rule base by the following algorithm:

Algorithm 3.1 *(Handle-Retrieve)*

```
for each entry in rangetable_query
    for each retrieve – lock on entry
        for each varnode which references entry
            if attribute number of varnode is locked
                replace varnode with ruletargetlist
                add rangetable entries as needed
                add rulequalifications
            else
                do nothing
```

If the query happens to be a *replace, append* or *delete*, then the following algorithm needs to be run as well :

Algorithm 3.2 *(Handle-Updates)*

```
for each rule defined on result_relation {
    replace references to current and new in the action clause
    with the appropriate attribute from targetlist_query
    qual_userquery = qual_userquery    and qual_rule
}
```

The sole reason for having *two* algorithms rather than *one* algorithm is that the structure of a 'Retrieve" parse tree is radically different from that of the other three types of queries (append, replace and delete), and thus the manipulation routines need to be different. The structural difference between the retrieve parse tree and the other parse trees is purely historical, and there

is no other reason why a future implementation should not be able to merge algorithms 3.1 and 3.2 into a single algorithm.

4 Examples and Timings

In this section, we show some examples of how QRS rules may be used to implement some database functions that have been typically hardcoded into the DBMS. Where applicable, we will also discuss the performance of each of these rulebases as compared with more "traditional" ways of implementing these functions.

4.1 Simple Constraints

One possible use of the QMS would be to maintain the constraint that an update to Sam's salary is propagated to Joe's salary. The rule to achieve this is :

define rule equal-sal **is**
on replace to *emp.salary* **where** *emp.name* = *"sam"*
do replace *e* (*salary* = *new.salary*) **from** *e in emp*
where *e.name* = *"joe"*

thus, subsequently when the application submits a query that update's Sam's salary :

replace e (*salary* = 7000) **from** e *in emp*
where $e.name$ = *"sam"*

the rule would be triggered, and the QRS produces two queries :

replace e (*salary* = 7000) **from** e *in emp*
where $e.name$ = *"sam"*

replace e (*salary* = 7000) **from** e *in emp*
where $e.name$ = *"joe"* **and** $emp.name$ = *"sam"*

In the second query, the additional clause ($emp.name$ = *"sam"*) is necessary in case there is no employee called "sam". Alternatively, if the programmer wishes to maintain the constraint from within the application he would have to submit the following two queries :

begin transaction
replace e (*salary* = 7000) **from** e *in emp*
where $e.name$ = *"joe"*
replace *emp* (*salary* = $e.salary$) **from** e *in emp*
where $emp.name$ = *"joe"* **and** $e.name$ = *"sam"*
end transaction

Intuitively, the rule-based method of maintaining the constraint performs at least as well as the application performing maintaining the constraint because of the following reasons :

1. **Number of Commands**

   QRS performs both updates in a single command, whereas, under all

circumstances, the application requires two commands to maintain the constraint. Thus, in this example, QRS eliminates the overhead of *one* command "commit". There is no way to avoid the extra command "commit" overhead when submitting two separate queries even if they are in the same transaction because by definition each user command can see the effect of the prior command. so the "commit" processing that takes place between any two commands is necessary. In contrast. QRS passes values between the user command and the rewrite system via "current" and "new". This does not require the user command to commit before the rule begins execution since they are logically part of the same command and the rules do not need to see the effect of the user command.

2. **Number of Messages**

   In order to enforce the constraint without rules, the application has to send two queries. By enforcing the constraint via the QRS, the application has to submit only one query. This reduces the number of messages between the application and the DBMS by at least half.

3. **Complexity of Queries**

Query Number	Rewrite Time (secs)	Query Time (secs)
User	0.0000	1.50
QRS	0.0078	0.48

Table 1: Performance of QRS vs Application Maintained Constraints

The application based method requires the use of a real join (Q2) whereas the rule-based method requires only an existential join because it substitutes the value of the new tuple into the query (Q3).

This intuition is borne out by the following statistics :

As we can see. the cost of maintaining this simple constraint from the application costs 1.5 secs. whereas the cost of maintaining it in QRS costs merely 0.48 secs.

4.2 Query Modification Views

POSTGRES views are identical to those described in [STON75] except for one aspect : View update semantics are "user programmable". As noted in [STON90], current relational view systems have certain limitations on the manner in which they may be updated. In particular, it seems particularly difficult to design an update policy in the absence of "view-specific" knowledge. There have been previous efforts to overcome these shortcomings,

though most, like [MEDE85] attempt to translate functional dependencies into view update strategies and completely fail when such functional dependencies do not exist. In contrast, since the update semantics in the POSTGRES view system are completely user definable, no functional dependencies are required.

In POSTGRES, this "Query Modification View System" is easily provided by translating a view definition into a simple set of QRS rules. For example, if we wanted to provide the canonical *toy* view of the *emp* relation, we would type:

```
define view toy ( emp.name, emp.salary )
   where emp.dept = "toy"
```

This view definition would then be automatically transformed by a view-compiler which we have implemented to produce the following rules:

```
define rewrite rule toy_retrieve is
on retrieve to toy
do instead
   retrieve ( emp.oid, emp.name, emp.salary )
       where emp.dept = "toy"


define rewrite rule toy_replace is
on replace to toy
do instead
   replace emp ( name = new.name , salary = new.salary )
```

Query	Execution Time	QRS overhead
retrieve (toy.name,toy.salary)	0.5001	0.0078
retrieve (emp.name,emp.salary) where emp.dept = "toy"	0.4923	0.0000

Table 2: Performance of View vs. Normal Relations (50% selectivity)

```
        where emp.oid = current.oid

define rewrite rule toy_delete is
on delete to toy
do instead
   delete emp where emp.oid = current.oid

define rewrite rule toy_append is
on append to toy
do instead
   append emp ( name = new.name, salary = new.salary,
                dept = "toy" )
```

Of course, any of the above rules can be subsequently replaced by a user defined rule so that the view-update semantics are exactly as desired by the user. Since POSTGRES does not already have a hard-wired view maintenance system, we are unable to provide a comparison between this new rule based method and the more traditional view system. We do however provide a comparison between queries to the view and queries to the actual relation in Table 2

The overhead in using QRS to maintain this simple view is mostly due to the recreation of rule parse trees from the "flattened" rule parse trees that

are stored in the *rewrite-rule* catalog. Caching of the structural rather than the flattened form would require major changes to the internals of POSTGRES. but would greatly reduce the QRS overhead. However, since the overhead is already fairly low (0.0078 secs), and stays constant regardless of the complexity of the query. it doesn't seem worth the extra effort to cache the "unflattened" rule structures.

4.3 Materialized Views

An alternate method of view maintenance is *immediate view maintenance* which provides the user with *materialized views* [BLAK86]. This kind of view is just as easily provided by providing a simple set of QRS rules.

For example. assuming that we materialize a view which consists of the ids, names and salaries of all employees who work in the toy department by submitting the query :

```
retrieve into toy ( emp.name, emp.salary, emp.id )
where emp.dept = ''toy''
```

The following rule definition will ensure that updates to the "emp" relation are automatically propagated to the "toy" view.

```
define rewrite rule mat_app is
```

```
on append to emp where new.dept = ''toy''
do append toy ( emp.name, emp.salary , emp.id )

define rewrite rule mat_rep1 is
on replace to emp where new.dept = ''toy''
      and current.dept != ''toy''
do append toy ( id = new.id, name = new.name,
                salary = new.salary )

define rewrite rule mat_rep2 is
on replace to emp where current.dept = ''toy''
do replace toy ( id = new.id, name = new.name,
                 salary = new.salary )

define rewrite rule mat_del is
on delete to emp where current.dept = ''toy''
do delete toy where emp.id = toy.id
```

A similar set of rules (which we will not present here) is required to ensure that updates from the "toy" view are propagated to the "emp" relation.

4.4 Procedure Rule

In POSTGRES, database procedures are known as POSTQUEL functions, and have been implemented as follows :

1. an incoming Postquel function definition similar to the following is received:

```
define postquel function DEPT(emp) returns dept  is
retrieve ( dept.all ) where dept.name = current.dept
```

2. Define a new attribute *DEPT* in emp. of type *dept* corresponding to the function name and the function return type as specified in the function definition.

3. Define the following rule by extracting the relevant information from the function body.

```
define rewrite rule dept is
on retrieve to emp.dept do instead
retrieve ( dept.all ) where dept.name = current.dept
```

5 Future Extensions

In this section, we describe various enhancements that could be made to the QRS, but were unable to implement at the current time. The list of enhancements is ordered by potential usefulness, rather than ease of implementation. Unfortunately, none of them are easily implementable.

5.1 Compiled Rules

As proposed in [STON86]. queries in Postgres could be compiled and stored in a compiled-plan catalog and plan invalidation would take place whenever the DBMS decided was necessary. Since rewriting of queries takes place before plans are stored. the stored plans will contain the rule-modified queries rather than just the original user query. This has the effect of reducing the runtime overhead of the QRS to zero. The only additional overhead might be the possibly greater occurence of plan invalidation because more complex queries are being run.

5.2 Parameterized Rules

To implement TP1 via database procedures, we would like to be able to define this POSTQUEL function :

```
define postquel function TP1 (customer,$1) {
     replace current ( balance = current.balance + $1 )
     replace branch ( balance = current.balance + $1 )
          where current.branch = branch.name
     retrieve ( b = branch.balance, c = customer.balance )
          where customer.name = current.name
          and current.branch = branch.name
}
```

where *customer* and *branch* are the tables where the customer's and branch's accounts are stored respectively. This function takes the value given in the parameter $1 and updates the customer and branch accounts with it. A positive value in $1 corresponds to a deposit and a negative value corresponds to a withdrawal from the account. Having defined this function, we could deposit $500 to Sam's account by running the following query:

```
retrieve ( customer.TP1(500).all ) where customer.name = "Sam"
```

Unfortunately, POSTQUEL functions are implemented as rules, and rules cannot have explicit parameters, so the above function cannot currently be defined as stated because it is parameterized. This example demonstrates that there are some functions where explicit parameters are very much desirable, and that we should explore parameterized rules.

5.3 Pseudo Recursive Rules/Procedures

As mentioned in Section 2.3, the rewrite system is inherently unable to process all *truly recursive* and most *pseudo recursive* rules. There is however, a subclass of "pseudo recursive" rules where $qual_{event} \cap qual_{action} \cap$

$qual_{userquery} = \emptyset$ that the rewrite system *should* be able to process. A "loop detector" could be written that would algebraically evaluate the qualifications to determine whether there was any recursion. Unfortunately, adding a "loop detector" provides little additional functionality to the user, since *truly recursive* rules still could not be processed. Furthermore, constructing such a "loop detector" is non trivial and the cost of algebraic evaluations is potentially quite high. Because of these considerations, in our current implementation both *pseudo* and *truly recursive* rules are disallowed by asserting that there are no recursive threads in any invocation of a rule.

6 Conclusion

In this paper, we have provided a detailed description of a Query Rewrite mechanism that will support the full syntax of PRS2 as described in [STON90]. The working examples in Section 4 show that the full spectrum of applications described in [STON90] are possible and have in fact been implemented in POSTGRES.

The timings provided in Section 4 further shows that such a system runs at least as fast as having the rules hardcoded into the application program.

Preliminary analysis shows that in the case of views, versions and "special" procedures, the Query Rewrite System (QRS) should outperform the Tuple Level System (TLS). However, it is not possible at this time to verify this by doing performance benchmarks as the TLS does not yet provide these functions.

Thus, at the time of writing, the functionality of the QRS is a superset rather than a subset of the TLS, even though in theory the reverse should be true. This may be attributed to the fact a QRS system is easily and efficiently implemented as a self contained module, whereas the TLS implementation, which requires hooks into many levels of the executor and access methods, is easily bogged down by the myriad interfaces required. This suggests that even though there are cases where the TLS could easily outperform the QRS, a database implementor should consider doing QRS first because the amount of functionality derived per man hour spent is greater.

References

[BLAK86] J.A. Blakeley, P. Larson, and F. W. Tompa. Efficiently updating materialized views. In *ACM-SIGMOD Conference on Management of Data*, May 1986.

[HANS89] Eric Hanson. An initial report on the design of ariel. In *Sigmod Record*, March 1989.

[MEDE85] Claudia Bauzer Medeiros and Frank Wm. Tompa. Understanding the implications of view update policies. In *Proceedings of the 11th VLDB Conference*, 1985.

[STON75] Michael R. Stonebraker. Implementation of integrity constraints and views by query modification. In *ACM-SIGMOD Conference on Management of Data*, June 1975.

[STON86] Michael R. Stonebraker and Lawrence A. Rowe. The design of postgres. In *Proceedings of the 1986 ACM-SIGMOD Conference*, Washington. D.C., June 1986.

[STON90] Michael R. Stonebraker. A. Jhingran. J. Goh. and S. Potiamanos. On rules. procedures. caching. and views in database systems. In *ACM-SIGMOD Conference on Management of Data*, 1990. to appear in ACM-SIGMOD Conference on Management of Data, 1990.

[WIDO90] Jennifer Widom and S. Finkelstein. A syntax and semantics for set-oriented production rules in relational data bases. In *Sigmod Record*. March 1990.