IMPROVING RULE PROCESSING IN POSTGRES DATABASE MANAGEMENT SYSTEM

Kyongsok Kim
Department of Computer Science
North Dakota State University
Fargo, ND. 58105-5075, U. S. A.

Abstract

This paper suggests methods to improve the performance of the rule subsystem of a next-generation relational database management system, called Postgres. Specifically, it will be shown that we can decompose some rules to get further optimization. Then, we will discuss how we can handle the situation when some fields are changed from indexable to nonindexable or vice versa. Finally, it is suggested that a new command be provided so that we can change indexable characteristics of more than one field at a time.

Keywords: decomposition, rule processing, early or late evaluation, source and target field, read- and write-set tag, indexable characteristics, optimization, random and priority semantics, mutually disjoint conditions

In Section 1, Postgres rule processing will be introduced. Then, in Sections 2 and 3, decomposing rules either by splitting their conditions or by splitting their target fields will be discussed, respectively, and some general comments about rule decomposition will be made in Section 4. We will discuss how we can change indexable characteristics of fields in Section 5. Finally, conclusions and recommendations will be given in Section 6.

1. Introduction

The design of a next-generation relational database management system, called Postgres (POST inGRES), that is the successor to the Ingres relational database management system, is described in [2]. One of the goals of this new database management system is to support rule processing within the database management system and therefore not only data values but also rules are stored within the database. Users can insert, retrieve, change, or delete rules just as they handle ordinary data values. Rule processing is handled by a rule subsystem of Postgres.

This Section basically paraphrases Postgres rule processing explained in [2, 3]. In Subsection 1.1, Postgres rule subsystem will be introduced. In Subsection 1.2, restrictions on choosing early versus late evaluation of rules will be discussed, and then, in Subsection 1.3, the Postgres implementation of those restrictions will be explained.

1.1. Postgres Rule Subsystem

The Postgres rule semantics will be described below using an example which is slightly modified from the one given in [3]. Consider the following EMP schema which has five fields:

EMP (employees) schema:

SSN	(9 characters,	indexable),
NAME	(20 characters,	indexable),
DEPT	(5 characters,	indexable),
SALARY	(integer,	nonindexable), and
BONUS	(integer,	nonindexable)

"Indexable" or "nonindexable" describes the "indexable characteristics" of fields. "Indexable" indicates that we may build an index using that field (possibly combined with other field(s)) as an indexing field; "nonindexable" indicates that, currently, we may not build an index using that field

as an indexing field. In Section 4, we will discuss how we should proceed when we want to build an index using a currently nonindexable field or when we want to change a currently indexable field to a nonindexable one.

The following replace command sets Kim's SALARY to Jane's SALARY.

```
replace EMPKIM (SALARY = EMPJANE.SALARY)
    from    EMPKIM in EMP,
            EMPJANE in EMP
    where   EMPKIM.NAME = "Kim" and
            EMPJANE.NAME = "Jane"
```

Now if we prefix "define rule Jane-to-Kim is alway" to the above replace command as shown below, the above replace command becomes a rule:

```
define rule Jane-to-Kim is always
replace EMPKIM (SALARY = EMPJANE.SALARY)
    from    EMPKIM in EMP,
            EMPJANE in EMP
    where   EMPKIM.NAME = "Kim" and
            EMPJANE.NAME = "Jane"
```

The above rule says that we define a rule, called Jane-to-Kim, which sets Kim's SALARY to that of Jane's. Its semantics is that, whenever Kim's SALARY field is retrieved, it will be the same as Jane's SALARY. One method to support this rule is that, whenever Jane's SALARY is changed, Kim's SALARY will be changed immediately. An alternative is to evaluate Kim's SALARY when it is actually retrieved. The former is called "early evaluation" and the latter is called "late (or lazy) evaluation".

If Jane's SALARY is updated frequently but Kim's SALARY is retrieved infrequently, we had better choose late evaluation, since we need to evaluate Kim's SALARY only when it is requested. In contrast, if Jane's SALARY is updated infrequently but Kim's SALARY is retrieved frequently, we had better adopt early evaluation, since we need to evaluate Kim's SALARY only when Jane's SALARY is updated. Therefore, in general, the time of evaluating rules has much effect on performance.

1.2. Restrictions on Choosing Early versus Late Evaluation of Rules

Whenever possible, Postgres tries to make a "correct" choice between early and late evaluations to optimize the rule processing. However, there are some restrictions which prevent Postgres from arbitrarily choosing one of the two evaluation methods. The first restriction is concerned about what (i.e., indexable or nonindexable) fields can or cannot be "written" by what (i.e., early or late) rules; the second restriction is concerned about what fields can or cannot be "read" by what rules. Now let's consider each of these two restrictions more carefully.

First, if some field is written by late rules, we cannot build an index using that field, since, sometimes, that field may not have the correct or up-to-date value. In contrast, we can build an index using some field which is written by early rules, since that field always has the correct or up-to-date value. Therefore early rules can write both indexable and nonindexable fields, whereas late rules can write nonindexable fields but not indexable fields.

Second, if we should mix early and late rules without any restriction, we will encounter problems. Let's consider an example which is slightly modified from the one given in [3]. Suppose that we have the Jane-to-Kim rule defined above and also the following rule:

```
define rule Mary-to-Jane is always
replace EMPJANE (SALARY =
            EMPMARY.SALARY)
    from    EMPJANE in EMP,
            EMPMARY in EMP
    where   EMPJANE.NAME = "Jane" and
            EMPMARY.NAME = "Mary"
```

Suppose that a) the Jane-to-Kim rule is evaluated early, b) the Mary-to-Jane rule is evaluated late, c) Mary has just received a SALARY adjustment, and d) Kim's SALARY is retrieved after Mary's salary is adjusted but before Jane's SALARY is retrieved. The Mary-to-Jane rule is a late rule and therefore Jane's SALARY will not be adjusted until somebody requests it. Since we assume that nobody has requested Jane's SALARY yet, Jane's SALARY still has the old value. Now, if we retrieve Kim's SALARY, we will see the old SALARY, not the new SALARY which should be equal to Mary's SALARY. From this example, we notice that we should not allow arbitrary mixing of early and late rules. To avoid this type of problem, Postgres ensures that no late rule writes any data item read by an early rule.

Since indexable fields can be written by early rules only, the values of indexable fields are always up-to-date and, therefore, any (early or late) rule can read indexable fields without any problem. In contrast, nonindexable fields can be written by early or late rules and, therefore, the values of nonindexable fields may or may not be up-to-date in general. The result is that, although late rules can read nonindexable fields without any problem, early rules should not be allowed to read nonindexable fields. (Actually this requirement seems somewhat too restrictive since some nonindexable fields are written by early rules and therefore can be read by early rules safely. A more complex mechanism may be able to lift this somewhat too restrictive requirement.) Therefore, late rules can read indexable and nonindexable fields, whereas early rules can read indexable fields but not nonindexable fields.

The above discussion can be summarized as follows: the first restriction says that late rules cannot write indexable fields and the second says that early rules cannot read nonindexable fields.

1.3. Implementation of Restrictions to Choosing Early versus Late Evaluation of Rule in Postgres

As we saw above, Postgres must make sure that the above two restrictions hold when processing rules. Let's see how Postgres implementation deals with these two restrictions. First, every field of a Postgres relation is labeled as either "indexable" or "nonindexable". Second, for each rule, we have two sets (or lists) of field names, read-set and write-set. A write-set contains those fields to the left of an assignment sign (those fields will be referred to as "target fields" in this paper). A read-set contains fields appearing to the right of an assignment sign (those fields will be referred to as "source fields") and fields in the conditions (i.e., predicates). In other words, a write-set contains target fields and a read-set contains source fields and those fields in conditions. For example, in the Jane-to-Kim rule, the write-set is

{SALARY},

and the read-set is

{SALARY, NAME}.

A rule whose read-set contains only indexable (or nonindexable) fields is tagged {read I} (or {read NI}), which will be referred to as the "read-set tag" of the rule in this paper. If a read-set contains a mixture of indexable and nonindexable fields, we tag the rule {read I and NI}. Similarly, based on the characteristics of write-set of a rule, we can tag the rule {write I}, {write NI}, or {write I and NI}, which will be referred to as the "write-set tag" of the rule. Both read-set and write-set tags of a rule will be collectively called "read-and-write-set tag". There are three read-set tags and three write-set tags, giving nine possible read-and-write-set tags.

Due to the first restriction, a late evaluation of a rule is not allowed when its write-set tag is {write I} or {write I and NI} (See Table 1). Due to the second restriction, an early evaluation of a rule is not allowed when its read-set tag is {read I and NI} or {read NI} (See Table 1). The result is that, out of nine possible cases, four cases are not permitted at all, two cases allow only early evaluation, two cases allow only late evaluation, and only one case allows us to evaluate early or late (See Table 1). This process derives Table 2 in [3], which summarizes the allowable execution times of rules for each combination of read-set and write-set tags. This is how Postgres implementation deals with the two restrictions mentioned in Subsection 1.2. Postgres tries to optimize the early versus late execution of rules which can be evaluated either early or late.

read-set \ write-set	I	I and NI	NI
I	early, late	early, late	early, late
I and NI	early, late	early, late	early, late
NI	early, late	early, late	early, late

Table 1. The process of deriving Table 2 below, Time of Rule Awakening.
/: not permitted due to the first restriction
(late rules cannot write indexable fields)
\: not permitted due to the second restriction
(early rules cannot read nonindexable fields)
O: allowable

read-set \ write-set	I	I and NI	NI
I	early	early	early or late
I and NI	not permitted	not permitted	late
NI	not permitted	not permitted	late

Table 2. Time of Rule awakening (slightly modified from the one given in [3]).

Based on Table 1, when the read-set tag is {read I and NI}, it can be treated as if it were {read NI}. Likewise, when the write-set tag is {write I and NI}, it can be treated as if it were {write I}. The result is that we can reduce the 3 by 3 table to a new 2 by 2 table.

Now let's turn to decomposing a rule into two or more new rules to facilitate further optimization.

2. Decomposing a Rule by Splitting its Conditions

In this Section, we will discuss decomposing a rule by splitting its conditions, when its read-set is a mixture of indexable and nonindexable fields. Specifically, we will consider three decomposition methods to make a further optimization in rule processing. Among the three methods, the two methods utilizing priority semantics

(Subsection 2.2) or mutually disjoint conditions (Subsection 2.3) outperform the method utilizing random semantics (Subsection 2.1).

2.1. Decomposition Utilizing Random Semantics

When two or more conditions in the where clause of a rule are connected by an "or" operator, we can decompose a rule into a set of two or more new rules. For example, consider the following rule:

```
define rule Adm1 is always
replace EMP (BONUS = 10000)
    where   EMP.NAME = "Jane" or
            EMP.SALARY > 40000
```

As it is, time of awakening Adm1 rule must be late, since the read-set contains both indexable (NAME) and nonindexable (SALARY) fields and the write-set contains only nonindexable field (BONUS). However, by decomposing the rule into two new rules, Adm11 and Adm12, we can facilitate further optimization:

```
define rule Adm11 is always
replace EMP (BONUS = 10000)
    where   EMP.NAME = "Jane"
    priority = 0

define rule Adm12 is always
replace EMP (BONUS = 10000)
    where   EMP.SALARY > 40000
    priority = 0
```

The effect of having two new rules, Adm11 and Adm12, is the same as that of having one original rule, Adm1. You may wonder why we put the equal priority to the two new rules. If Jane's SALARY is greater than 40,000, both Adm11 and Adm12 rules will attempt to update the same field (BONUS) even though either rule will produce the correct result. By putting the equal priority, we utilize the random semantics of Postgres, since, if multiple rules have the same priority, Postgres uses random semantics for conflicting rules and returns the result specified by any one of them.

Now let's investigate each of the new rules. The read-set of rule Adm11 contains an indexable field (NAME) and the write-set contains a nonindexable field (BONUS). Therefore time of awakening rule Adm11 can be either early or late, which facilitates further optimization by Postgres. Note that the original rule did not allow this optimization. Time of awakening rule Adm12 still remains late.

You may wonder what is the possible gain in decomposing in the above example. Suppose that Jane's SALARY is less than 40,000 and Jane's BONUS is retrieved quite frequently. Since the original Adm1 rule must be evaluated late, we have to evaluate Jane's BONUS every time it is retrieved. However, with a new rule Adm11, Postgres can decide to evaluate this rule early. Then Jane's BONUS will be evaluated just once, however frequently her BONUS is retrieved.

In general, if the where clause contains two or more conditions connected by an "or" operator as shown below, we can decompose the rule into a set of two or more new rules and, if three conditions explained below are met in addition, we will get some new rule(s) which may be evaluated early or late, facilitating further optimization by Postgres.

```
an original rule:
always replace ... where Cond-1 or Cond-2 or ...

->
a new rule 1:
always replace ... where Cond-1

a new rule 2:
always replace ... where Cond-2
    ....
```

Now consider the read-and-write-set tag of each of new rules and its awakening time after decomposing a rule whose read-and-write-set tag is {read I and NI, write NI}. In general, there are five possible cases as shown below. The above example where we decomposed Adm1 rule into Adm11 and Adm12 rules falls into case 1.

case	read-and-write-set tags	time of rule awakening
0	{read I/NI, write NI}	late
1	{read I, write NI}, and {read NI, write NI}	early or late late
2	{read I, write NI}, and {read I/NI, write NI}	early or late late
3	{read I/NI, write NI}, and {read NI, write NI}	late late
4	{read I, write NI}, {read I/NI, write NI}, and {read NI, write NI}	early or late late late

Table 3. Read-and-write-set Tags of, and Time of Awakening, New Rules after Decomposing a Rule by splitting its Conditions.

Among those five cases, only cases 1, 2, and 4 produce new rule(s) which can be evaluated early or late. It needs more investigation to analyze whether we will get any gain by decomposing a rule even in cases 0 and 3.

Now let's consider when we can get, after decomposition, at least one new rule which can be evaluated early or late. The read-and-write-set tag of such a new rule is {read I, write NI}. Therefore we need the following three conditions: a) the read-and-write-set tag of the original rule must be {read I and NI, write NI}, b) all target fields are indexable, and c) there is at least one condition in the where clause that includes only indexable fields or constants. If the three conditions are met, we can get, after decomposition, at least one new rule which can be evaluated early or late.

Although the decomposition method in this Subsection works by utilizing the random semantics of Postgres, some people may not feel comfortable. For example, in Adm11 and Adm12 rules above, if Jane's SALARY is greater than 40,000, they may be afraid that both rules "might" adjust Jane's BONUS, which is not the case.

Now let's consider how we can improve the decomposition method which relies on random semantics. In the next two Subsections, 2.2 and 2.3, we will consider two alternative "improved" decomposition methods which do not rely on random semantics.

2.2. Decomposition Utilizing Priority Semantics

One alternative decomposition method is to assign different priority to each of new rules. As shown below, we can assign priority 1 and 0 to Adm13 and Adm14 rules, respectively. We intentionally assigned higher priority to Adm13 rule, since we want Adm13 rule, which can be evaluated early or late, to cover as many tuples as possible. That is, if Jane's SALARY is greater than 40,000, her BONUS will be evaluated by rule Adm13, which can be evaluated early or late, not by rule Adm14 which must be evaluated late. Although the gain here seems negligible, in general, the gain will be significant when the intersection of two conditions covers many tuples.

```
define rule Adm13 is always
replace EMP (BONUS = 10000)
    where   EMP.NAME = "Jane"
    priority = 1

define rule Adm14 is always
replace EMP (BONUS = 10000)
    where   EMP.SALARY > 40000
    priority = 0
```

We can generalize this idea. Suppose that we have an original rule that meets the three conditions mentioned at the end of Subsection 2.1. Then we can decompose the rule into two new rules. One rule with the higher priority includes all conditions having only indexable fields or constants so that it can be evaluated early or late. The other rule with the lower priority contains the conditions not included in the first rule.

2.3. Decomposition Utilizing Mutually Disjoint Conditions (or Predicates)

Another alternative decomposition method is to make the conditions (or predicates) in new rules mutually disjoint. Consider again the Adm1 rule. We can decompose it as follows:

```
define rule Adm15 is always
replace EMP (BONUS = 10000)
    where   EMP.NAME = "Jane"


define rule Adm16 is always
replace EMP (BONUS = 10000)
    where   (not (EMP.NAME= "Jane")) and
            EMP.SALARY > 40000
```

Note that a) the negation of the condition in rule Adm15 is included in the where clause of Adm16 rule, and b) we do not use priorities. Now the predicates in rules Adm15 and Adm16 are mutually disjoint and therefore we do not need to use priority any more. Rule Adm15 can be evaluated early or late whereas rule Adm16 remains as a late rule. Here we also intentionally added the negation of the condition in rule Adm15 to rule Adm16, not vice versa, since we want rule Adm15, which can be evaluated early or late, to cover as many tuples as possible. The reasoning here is exactly the same as the one above in assigning different priorities in Subsection 2.2.

We can generalize this idea. Suppose that we have an original rule that meets the three conditions mentioned at the end of Subsection 2.1. Then we can decompose the rule into two new rules. One rule includes all conditions having only indexable fields or constants so that it can be evaluated early or late. The other rule contains negations of those conditions included in the first rule, in addition to the conditions not included in the first rule.

Decomposition methods utilizing priority semantics or mutually disjoint conditions seem to give better performance than the one utilizing random semantics. It needs more investigation to check which of the two will perform better.

3. Decomposing a Rule by Splitting its Target Fields

In Subsection 3.1, we will discuss decomposing a rule by splitting its target fields, when its write-set is a mixture of indexable and nonindexable fields, and then, in Subsection 3.2, we will discuss a possible side-effect of this decomposition.

3.1. Decomposing a Rule by Splitting its Target Fields

Now consider a rule whose read-set tag is {read I} and write-set tag is {write I and NI}. This occurs when a) all source fields and those fields in the conditions are either indexable fields or constants and, b) there are two or more target fields, among which at least one field is indexable and at least one field is nonindexable. We can always decompose such a rule as follows:

an original rule:
always replace (field1 = ..., field2 = ..., ...)
    where Cond(s)

->
a new rule 1:
always replace (field1 = ...) where Cond(s)

a new rule 2:
always replace (field2 = ...) where Cond(s)
    ....

Let's consider the read-and-write-set tag of each of new rules after decomposing a rule whose read-and-write-set tag is {read I, write I and NI}. In general, the tag of each of the new rules will be either {read I, write I} or {read I, write NI} since, after decomposition, each new rule will have only one field in the write-set which must be either indexable or nonindexable.

read-and-write-set tags	time of rule awakening
{read I, write I}, and {read I, write NI}	early early or late

Table 4. Read-and-write-set Tags of, and Time of Awakening, New Rules after Decomposing a Rule by splitting its Target Fields.

In general, we can decompose such a rule into two new rules. One rule, which must be evaluated early, includes all indexable target fields. The other rule, which can be evaluated either early or late, includes all nonindexable target fields. This latter rule facilitates further optimization by Postgres.

3.2. A Possible Side-effect of Decomposing a Rule by Splitting Target Fields

In the above example, the original rule updates all target fields at a time. In contrast, each of new rules updates only some of the target field(s) in the original rule at a time. Therefore the number of update operations by the original rule and the number of update operations by the new rules may not be the same in general. If the exact number of update operations affects, for example, some user's actions, other rules, audit log, etc., there is a side-effect of decomposing a rule by splitting its target fields.

In general, once we allow a database management system to evaluate some rules early or late, it seems that we are not supposed to rely on "how" the evaluations of rules will be done "internally". Instead, as long as the rules work "somehow", we may have to feel satisfied. In other words, the internal rule processing should be transparent to the users. This kind of problem seems somewhat philosophical. Nevertheless, it seems worthwhile to investigate more about possible multi-rule interactions caused by a rule decomposition.

4. Some General Comments about Rule Decomposition

Now, we will discuss some comments which are common to decomposing rules either by splitting their conditions or by splitting their target fields. In Subsection 4.1, it will be shown that rules that are not originally permissible cannot be decomposed. Then, in Subsections 4.2 and 4.3, we will discuss whether we need to decompose a rule even if the read-and-write-set tags of, or time of awakening, those new rules remain unchanged, respectively.

4.1. Rules that are not Originally Permissible Cannot be Decomposed

As you can see in Table 2, there are five possible combinations where the read-set and/or write-set is a mixture of indexable and nonindexable fields. Among those, only two combinations, {read I/NI, write NI} and {read I, write I/NI}, allow decomposition as shown above in Sections 2 and 3, respectively. Rules of the other three combinations (i.e., {read I/NI, write I/NI}, {read I/NI, write I}, and {read NI, write I/NI}) are not permitted; furthermore, even if they are decomposed, at least one of the new rules is not permitted, which invalidates the decomposition.

4.2. Do we need to Decompose a Rule even if the Read-and-write-set Tags of (and therefore Time of Awakening) those New Rules Remain the Same as Before?

In Subsection 2.1, we already saw that, after we decompose a rule by splitting conditions in its where clause, the read-and-write-set tags of those new rules may remain the same as before. For example, if a rule has two conditions, each of which includes both indexable and nonindexable fields, this rule can be decomposed into two new rules whose read-and-write-set tags remain {read I and NI, write NI} (See case 0 in Table 3).

If we have a rule whose read-set includes only indexable (or nonindexable) fields, we may decompose the rule by splitting conditions into new rules whose read-and-write-set tags remain {read I, write NI} (or {read NI, write NI}).

A similar situation can happen when we decompose a rule by splitting its target fields. For example, if all target fields are indexable (or nonindexable), we may decompose the rule by splitting its target fields into new rules whose read-and-write-set tags remain {read I, write I} (or {read I, write NI}).

It needs further investigation to analyze whether we will get any gain by decomposing rules even when the read-and-write-set tags of those new rules remain the same as before.

4.3. Do we need to Decompose a Rule even if Time of Awakening those New Rules Remain the Same as Before?

After decomposition, sometimes, time of awakening those new rules may remain the same as before, even though the read-and-write-set tags of some of new rules have been changed. Consider case 3 in Table 3. Even though we get one or more new rules whose read-and-write-set tag, {read NI, write NI}, is different from that of the original rule, {read I and NI, write NI}, the time of awakening those new rules remain the same as before (i.e., late).

If the read-and-write-set tags of new rules remain unchanged, of course, the time of awakening those new rules remain unchanged. Therefore, the cases discussed in Subsection 4.2 where read-and-write-set tags of new rules remain unchanged, can be considered as a proper subset of those cases where time of awakening new rules after decomposition remain unchanged.

It needs further investigation to analyze whether we will get any gain by decomposing rules even when the time of awakening those new rules remains the same as before.

5. Changing Indexable Characteristics of Fields

In Subsection 5.1, we will discuss what Postgres should do when a user wants to change the indexable characteristics of fields. Then, in Subsection 5.2, we suggest a new command be provided for changing indexable characteristics of more than one field at a time.

5.1. The Effects of Changing Indexable Characteristics of Fields

From time to time, we may want to change indexable characteristics of some fields from indexable to nonindexable or vice versa. For example, we may want to build a new secondary index using a currently nonindexable field as an indexing field, or we may choose not to maintain an index for some field any more. When such a change occurs, we need to take necessary actions.

In general, when indexable characteristics of some fields changes, there are too many possible transitions. By considering two actions separately, we can simplify the transition process. One action is concerned with the change in time of awakening the affected rules (i.e., those rules whose read-set and/or write-set contains that field). Time of awakening the rule may remain the same as before, or change, or the rule may not be permitted any more. The other action is concerned with whether index(es) should be destroyed.

First, consider the change in time of awakening the rules affected. Depending on i) the current time of awakening the rule, ii) whether or not the rule will still be permitted, and, if permitted, the new time of awakening the rule, we can consider six different transitions as summarized in Table 5 (a). Let's consider what Postgres should do for each of those six transitions.

a) When an early or late rule remains unchanged, we do not have anything to do.

b) When an early rule becomes a late rule, we may need to invalidate those values evaluated by the rule. The exact action seems to depend on the implementation.

c) When a late rule becomes an early rule, we need to evaluate those fields in the write-set of the rule and write them into database.

d) When an early rule becomes "not permissible", we need to invalidate (delete) the rule.

e) When a late rule becomes "not permissible", we need to evaluate those fields in the write-set of the rule, write them into database, and then invalidate (delete) the rule.

Second, consider the change of indexable characteristics of a field. There are only two transitions: from indexable to nonindexable and vice versa, as shown in Table 5 (b). Let's consider what Postgres should do for each of those two transitions.

a) When an indexable field becomes nonindexable, we

need to destroy indexes, if any, where the field was used as an indexing field.

b) When a nonindexable field becomes indexable, we don't need to anything. If a user wants to build an index, he/she (not Postgres) can do.

new time of awakening a rule \ current time of awakening a rule	early	late
early	o.k.	evaluate and write
late	invalidate current values	o.k.
not permissible	invalidate the rule	evaluate and write; invalidate the rule

(a) Actions taken due to change in time of awakening a rule.

indexable characteristics of a field		actions taken
current	new	
indexable	nonindexable	destroy indexes, if any
nonindexable	indexable	o.k.

(b) Actions taken due to change in indexable characteristics of each field.

Table 5. Actions taken when indexable characteristics of field(s) changes.

5.2. Suggestion: a New Command be Provided for Changing Indexable Characteristics of More than One Field at a time

If Postgres provides a command for changing indexable characteristics of only one field at a time, then we have some problem, when we want to change indexable characteristics of more than one field at a time. Consider the rule Adm1 in Subsection 2.1. Its read-set is {NAME, SALARY} and write-set is {BONUS}. Since NAME field is indexable and SALARY and BONUS fields are nonindexable, its read-and-write-set tag is {read I and NI, write NI}, and therefore it is a late rule. Suppose that we want to change both SALARY and BONUS fields at a time from nonindexable to indexable. Since we assumed that we can change indexable characteristics of one field at a time, there are two possible sequences of steps: a) changing SALARY field first, then BONUS field; or b) changing BONUS field first, then SALARY field. The problem is that, sometimes, these two sequences may give different results.

Consider what happens when we follow sequence a). When we make SALARY field indexable, the read-set tag changes from {read I and NI} to {read I} and the rule can be evaluated early or late (See Table 6). Then, when we make BONUS field indexable, the write-set tag changes from {write NI} to {write I} and the rule must be evaluated early. It seems fine. That is what we wanted originally.

Now consider what happens when we follow sequence b). When we make BONUS field indexable, the write-set tag changes from {write NI} to {write I} and the rule becomes "not permissible" (See Table 6). Then, we must delete (or drop) this rule. This is not what we wanted originally.

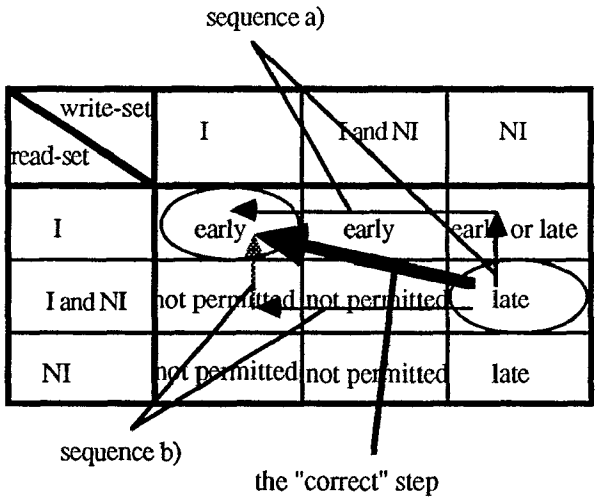

Table 6. Transition of Time of Rule Awakening when indexable characteristics of SALARY and BONUS fields changes from nonindexable to indexable.

In general, if indexable characteristics of n fields are to be changed, there are n! (n factorial) possible sequences of n steps. To "correctly" handle this situation, we need to

change all of them "in one step", not one by one in a sequence of n steps. In the above example, the read-and-write-set tag must be changed from {read I and NI, write NI} to {read I, write I} in one step, as shown in Table 6.

The format of the suggested command might look like:

```
change-indexable-characteristics
    relname1.fieldname1
        from indexable to nonindexable,
    relname2.fieldname2
        from nonindexable to indexable,
    ....
end-change-indexable-characteristics;
```

It will also be helpful to provide a flag so that we can see the possible effects of changing indexable characteristics of some fields (e.g., some rules must be invalidated, some indexes must be destroyed, etc.) without actually changing anything in the database. If some effects are not acceptable, we can simply give up doing so without affecting the database in any way. To "undo" the effects of changing indexable characteristics of fields may be too costly, especially when an index has already been destroyed.

6. Conclusions

We reviewed the rule processing of a next generation relational database management system, called Postgres, and suggested some methods to improve the performance of its rule subsystem by decomposing rules into a set of new rules. We can decompose a rule either by splitting its conditions or by splitting its target fields. When we split conditions, we can utilize random semantics, priority semantics, or mutually disjoint conditions. Since some of these new rules can sometimes be evaluated either early or late, we can get further optimization.

We also discussed what we should do when some fields are changed from nonindexable to indexable or vice versa. It was suggested to provide a command so that we can change indexable characteristics of more than one field at a time.

It needs more investigation to analyze whether we will get any gain by decomposing rules even when time of awakening, or read-and-write-set tags of, those new rules remain unchanged.

References

[1] The Postgres Reference Manual, University of California, 1988.

[2] Michael Stonebraker and Lawrence A. Rowe, "The design of Postgres", in Proceedings of ACM SIGMOD 86, International Conference of Management of Data, Washington, D.C., May 28-30, 1986, SIGMOD Record, Vol. 12, No. 2, June 1986.

[3] Michael Stonebraker, et. al., "The Postgres rule manager", IEEE Transactions on Software Engineering, Vol. 14, No. 7, July 1988.