SRC TR 87-217

Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms

by

T. Sellis, Chih-Chen Lin, L. Raschid

Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms[1]

Timos Sellis[2], Chih–Chen Lin

Department of Computer Science
and Systems Research Center

Louiqa Raschid

Department of Information Systems
School of Business and Management

University of Maryland
College Park, MD 20742

Abstract

It has been widely recognized that many future database applications, including engineering processes, manufacturing and communications, will require some kind of rule based reasoning. It is conceivable that large knowledge bases cannot, and perhaps should not, for space reasons, reside in main memory. In this paper we study methods for storing and manipulating large rule bases using relational database management systems. First, we provide a matching algorithm similar to the Rete Network used in OPS5, which can be used to efficiently identify applicable rules. The second contribution of this paper, is our proposal for concurrent execution strategies which surpass, in terms of performance, the sequential OPS5 execution algorithm. Since the problem of identifying applicable rules is the same as the problems of supporting triggers and materialized views in a conventional relational database system, our approach provides some new ideas for the solution of these problems as well. Finally, the proposed method is fully parallelizable, which makes its use even more attractive, as it can be used in parallel computing environments.

[1] This research was sponsored partially by the National Science Foundation under Grant CDR–85–00108 and by UMIACS.
[2] Also with University of Maryland Institute for Advanced Computer Studies (UMIACS).

1. Introduction

It has been widely recognized that many future database applications, including engineering processes, manufacturing and communications, will require some kind of rule based reasoning. It is conceivable that a large knowledge base cannot, and perhaps should not, for space reasons, reside in main memory. This is exactly the point where DataBase Management Systems (DBMS) come to play. However, applications such as the ones referenced above, require control mechanisms much more sophisticated than the ones current DBMS's can offer (simple value matching). For this reason a lot of research effort has been devoted to studying the support of more advanced control mechanisms in database environments, such as rules, deductive inference, recursion, and forward chaining, to name a few [KERS86,KERS87].

Commercial DBMS's have limited capabilities for supporting such mechanisms. For example, deductive rules can be "simulated" using views, though without allowing multiple or recursive rule definitions. Deductive inference can then be achieved through query modification [STON75]. In the case of multiple and recursive definitions new execution mechanisms need to be incorporated. Recent work in this area has provided a lot of results [CHAK86,BANC86,IOAN86,SELL88]. More general kinds of rule systems, such as production rule systems [HAYE85], are harder to incorporate because they require mechanisms to propagate updates to the database, in contrast to deduction which just retrieves data from the database.

Existing relational systems have some limited rule subsystems in the form of integrity control and protection subsystems. Updates are "filtered" and performed only if several user-defined constraints are met. In a general production rule system environment, updates to the database may trigger the firing of some rules, which in turn may perform several updates to the database, etc. This control mechanism introduces several sub-problems to be solved, such as, how to efficiently trap updates, how to process actions of rules that have been triggered, and what kind of low-level support is needed for all the above. The problem of supporting production systems efficiently in a database environment will be the focus of this paper.

The organization of the paper is as follows: Section 2 introduces the problem and surveys previous work in the area. Then in Section 3 we study the solution that the Artificial Intelligence (AI) community favors and discuss its advantages and disadvantages. Section 4 then looks at various ways of implementing production systems in a DBMS environment and compares them to the AI approach. In Section 5 we discuss execution strategies that allow for concurrent processing of qualifying rules and we conclude this paper in Section 6 with a summary and future research issues.

2. Production Rule Systems

Both the DBMS and AI community have been studying problems related to production rules, usually under different contexts. Hanson [HANS87] offers a very good discussion of past and recent research. In the following, we first describe the problem that is of interest and then present some of the basic approaches.

2.1. Rules

In the area of expert systems, a production system program is a collection of *Condition–Action* statements, called *productions* [FORG82] or *rules.* The condition part of a production is referred to as the LHS (left–hand side) of the production; similarly, the action part is called the RHS (right–hand side) of the production. Rules operate on data stored in a global database, called *working memory* (WM). A production system repeatedly performs the following operations, and in the sequence they are presented

Match
For each rule r, determine if LHS(r) is satisfied by the current WM contents. If so, add the qualifying rule to the *conflict set.*

Select
Select one rule out of the conflict set; if there is no such rule, halt.

Act
Perform the actions in the RHS of the selected rule. This will change the content of the WM and new rules may have to be fired.

The above procedure implies that two significant problems must be solved. First, one needs a fast way for performing the first step, i.e. finding qualifying rules. This may not be important in an environment with a few rules but becomes critical in the case of large rule bases and/or when secondary storage is used to store the WM elements. Second, the process of selecting one rule out of the conflict set may be very complicated, depending on the application. One may use user–defined priorities or, in general, order rules according to some static or dynamic criteria and then fire the rules in that order. Of course, the execution of a rule may cause other rules to fire, which may cause preemption of other rule executions, etc. Another way, which is of practical importance in a database environment, would be to allow all selected rules to execute in parallel and let the concurrency control manager of the DBMS take care of concurrent accesses to the same data by serializing updates. We discuss this problem further in Section 5. In the remaining of this section and the following two sections, we focus on the problem of efficient matching.

We should finally note that the problem of supporting production systems in a DBMS environment is of interest to conventional DBMS's as well. Productions can be thought of as triggers. Maintenance of materialized views also requires mechanisms to trap and propagate updates. It will become clear later on how the various approaches can be used to solve problems such as view maintenance.

2.2. The AI Way

The most representative approach to efficient matching has been the Rete Match Algorithm [FORG82] used in the OPS5 system [FORG81]. The Rete algorithm compiles the LHS condition elements into a binary discrimination network. Elements that are inserted to or deleted from the system are input into the discrimination network and flow through its nodes. Each node of the network stores tokens corresponding to WM elements that satisfy the network, i.e. the conjunction of the condition elements above that node.

Example 1: Suppose that we have a LHS condition of the form

$$C_1 \wedge C_2 \wedge C_3 \ \cdots \ \wedge C_n$$

Figure 1 shows the discrimination network built. The output from the network is all the applicable productions whose LHS is satisfied, i.e. the conflict set. In addition to that, the tokens that satisfy the above LHS's are also output. □

Notice that the above network is an inherently redundant storage structure since it stores a token for each WM element satisfying a rule condition and a single WM element could simultaneously satisfy several rule conditions. Moreover, tokens flow through the network in a serial way, each node examined at each step. We elaborate more on the Rete Match algorithm in

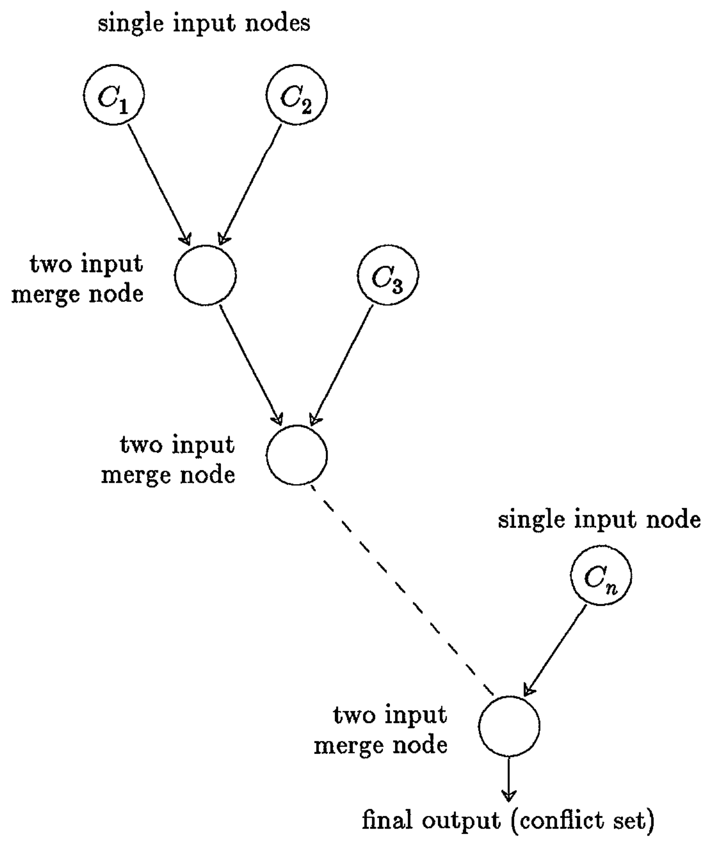

Figure 1: Example of a Discrimination Network

Section 3 in the context of its implementation in a DBMS environment.

One observation that can be easily made here is that LHS's are equivalent to retrieval operations in a DBMS context. Satisfying the LHS of a production is equivalent to executing a retrieval operation against the occurrences in a database. However, it is the usual case in a DBMS environment, to process retrievals only when a user issues a query to the system. In the case of production systems, the satisfiability of a condition is checked at the time an update is performed to the database; such operations have also appeared in database contexts and are discussed in the next sub–section.

2.3. The DB Way

Previous work relative to production systems has focused on database *triggers* and *alerters*. A trigger is a condition and an associated action to be executed if the database comes to a state that makes the condition true. An alerter is a trigger that sends a message to a user or an application program if its condition is met. Triggers have been studied by Eswaran in [ESWA76] in the context of concurrency control and authorization. In particular, this work has examined what kind of privileges that triggers should have and how they should relate to the transaction that forced them to fire. Perhaps the first systematic work has been the one by Buneman and Clemons [BUNE79]. They discuss two classes of triggers, *simple* and *complex*. Simple triggers basically account for single relation conditions while complex ones involve conditions that include multi–relation operations (e.g. joins). Buneman and Clemons put the problem in the context of supporting materialized views in a relational DBMS. The qualifications of the view definitions are used to make up the collection of conditions that must be monitored. They use *add* and *delete* triggers that are awakened when a tuple should be inserted or deleted from the view, respectively. The triggering mechanism they propose requires recomputing the view after each update. However, since recomputing the view is very expensive, they developed a method that checks if updates must be propagated based on the idea of *Readily Ignorable Updates* (RIU). An RIU is an update that can be determined in advance not to affect the view used as the condition of a rule. Similar work has been done recently on the same subject by Blakeley, Larson and Tompa [BLAK86a], and Blakeley, Coburn and Larson [BLAK86b].

Recently, Stonebraker proposed an extension to QUEL [STON76] commands to model triggers [STON85]. Stonebraker, Sellis and Hanson have studied the details of the implementation of such triggers [STON86a]. Triggers are formed by tagging any QUEL command with the keyword "ALWAYS". Such tagged commands conceptually appear to run indefinitely. To give an example drawn from [STON86a], assuming a relation `EMP(name,salary,age,dept)`, with the obvious meanings for its fields, a trigger that forces Mike's salary to always be equal to Sam's salary is expressed as follows:

```
range of E is EMP
replace ALWAYS EMP (salary = E.salary)
where   EMP.name = "Mike"
and     E.name = "Sam"
```

Whenever a command such as

```
replace EMP (salary = 1000) where EMP.name = "Sam"
```

is processed, the trigger should be awakened to update Mike's salary.

In general, the system maintains a collection of triggers. When a user update U is processed, the system must find all triggers that might have to be fired because of U. Of course, depending on the complexity of the algorithm that looks for satisfiable conditions, the system may awaken a trigger even when it should not (*false drops*). However, the penalty to be paid is just in processing time, the database will not become inconsistent. This is due to the semantics of "ALWAYS" commands.

Although it has been the case with the Rete algorithm that only one–tuple–at–a–time processing is feasible, database environments have always been based on the set–at–a–time concept. In [STON86a], Stonebraker, Sellis and Hanson, have suggested mechanisms to efficiently detect qualifying rules in a DBMS environment ("*rule indexing*"). The methods presented there can also be used for maintaining materialized views, since, as mentioned above, the qualifications of the views can be thought of as the conditions of rules that take actions to keep the views up–to–date. We briefly present here the methods described in [STON86a].

The two approaches taken, *Basic Locking* and *Predicate Indexing*, share the same properties with physical and predicate locking respectively [GRAY77] as used in concurrency control. Abstractly, a set of tuples is used to produce the result of some query and our goal is to be able to detect when a given update *conflicts* with this set. Hence, the similarity with the concurrency control problem.

In Basic Locking, all tuples used in processing a given condition or view qualification are marked with a special kind of marker, which is used to uniquely identify the condition. If an index is used for accessing the data tuples, these markers are set on data records *and* on the key interval inspected in the index. Index interval locks are required to deal correctly with insertion of new records (the *phantom problem* in concurrency control). If a new tuple is inserted in one of the relations used to produce the result of a procedure entry, then the collection of markers must be found for the new tuple. To ascertain what collection of cached entries are affected by the insertion of a tuple t, one first collects all the markers on t and then determines which of the corresponding conditions are really affected.

In Predicate Indexing, a data structure similar to a discrimination network is built. Such a structure allows for the efficient search and detection of conditions (LHS's) affected by the insertion of a specific tuple in the database. In [STON86a], it is suggested that a variation to R–trees

[GUTT84], R^+-trees [SELL87], are used for that reason. Using Predicate Indexing implies no special treatment of insertions to base relations, but a search of the whole tree is required whenever one asks for the conditions affected by an update.

Performance analysis results in [STON86a], show that it is not possible to choose one implementation to efficiently support any rule-based environment. Depending on the probability of updating base relations and the number of conditions that overlap (in the sense that their read sets share some tuples from base relations), the first or the second approach becomes more efficient. Analysis of these schemes and investigation of other extensions are a topic of current research.

Given this discussion we move next to present other ways of implementing production rule systems in database environments. The following sections show how ideas from the Rete algorithm can be tied together with relational DBMS technology to obtain better indexing schemes.

3. AI Indexing Techniques

In this section we present in more detail the most prominent AI technique for rule indexing and discuss its implementation in a DBMS environment.

3.1. The OPS5 Approach

As mentioned in the previous section, the most representative of all methods used in AI is the Rete Match Algorithm invented by Forgy [FORG82]. In OPS5, the database resides entirely in virtual memory, and does not persist after the execution of a program. An OPS5 rule consists of (1) the symbol **p**, (2) the name of the rule, (3) the LHS, (4) the symbol →, and (5) the RHS. Parentheses are used to enclose everything.

Example 2: The following are two rules [FORG82]

```
(p PlusOx
     (Goal    ↑ Type Simplify  ↑ Object <N>)
     (Expression  ↑ Name <N>  ↑ Arg1 0  ↑ Op +  ↑ Arg2 <X>)
→
     (modify 2  ↑ Op NIL  ↑ Arg1 NIL))
```

```
(p TimeOx
     (Goal   ↑ Type Simplify  ↑ Object <N>)
     (Expression  ↑ Name <N>  ↑ Arg1 0  ↑ Op *  ↑ Arg2 <X>)
→
     (modify 2  ↑ Op NIL  ↑ Arg2 NIL))
```

that can be used to simplify algebraic expressions. The effect of the **modify** statement is to write NIL into the "Op" and "Arg2" fields of the data item matching condition element number 2, that is the second term of the condition. □

Possible statements in a rule include, **modify**, for updating fields, **remove**, to remove data elements, **make**, to insert new data elements, and **call**, for calling general procedures. As it can seen in the above examples, joins are implemented through common variables (e.g. variable <N> above).

The Rete Match Algorithm is used in OPS5 to reduce the computation required to check for conditions that are satisfied. Forgy mentions in [FORG82] that the Rete Network has been introduced "to avoid iterating over working memory". Essentially, what is suggested is to evaluate the conditions of the various rules and monitor changes in the database in an efficient way. This is achieved by keeping all matches among working memory elements, in database terminology, all tuples satisfying selections or pairs of tuples satisfying joins. Given this information, OPS5 uses the Rete Network as shown in Figure 2. The descriptions of working memory

$$\begin{tikzcd}
\text{Changes to Working Memory} \arrow[d, "\textit{propagate}"] \\
\text{Rete Network} \arrow[d, "\textit{detect matches}"] \\
\text{Changes to Conflict Set}
\end{tikzcd}$$

Figure 2: OPS5 Function

(WM) changes that are propagated to the Rete Network are called *tokens.* Tokens are simply tuples tagged with a "+" or a "−" to show whether the tuple was inserted or deleted, respectively. It is assumed that modifications are treated as deletions followed by insertions. As shown in Figure 2, the algorithm maintains a *conflict set* which contains information on all applicable rules and the data elements (tuples) that cause these rules to fire.

Rule definitions are compiled and the discrimination network is produced. For example, Figure 3 illustrates the result of compiling the two rules of Example 2. There is a *root* node which receives all the tokens that are input to the network. *One–input* nodes are used to check single attribute conditions. That is conditions of the form

`attribute` *op* `constant`

where $op \in \{<,>,\leq,\geq,=,\neq\}$. Finally, *two–input* nodes are used to check joins of the form

`left-input.attribute` *op* `right-input.attribute`

Data elements are input through the root. Suppose a tuple *t* is input to the network of Figure 3. The one–input nodes are first checked to determine if it is a "Goal" or "Expression" tuple. If *t* does not meet either qualification, it is discarded. Otherwise it is propagated to the successors of the qualifying node of the network. In case a check is performed at a two–input node, and a matching value is not found at the corresponding join branch, the tuple is queued up at the network waiting for a future arrival of a matching tuple. When such a tuple comes through the network, the result of the join is propagated to the successors of the two–input node. Finally, if a token makes it all the way till the "bottom" of the Rete Network, a rule or set of rules have qualified and the system adds these rules to the conflict set, together with the token that caused the rule to become active.

Using the above algorithm, updates are treated incrementally and re–computation is avoided. The algorithm is very similar to the one suggested by Blakeley et al. [BLAK86a,b]. However, in this latter approach *all* materialized view results are checked if they are affected by a given update. Using the Rete Network, OPS5 avoids that by quickly discarding rule conditions that clearly cannot be affected.

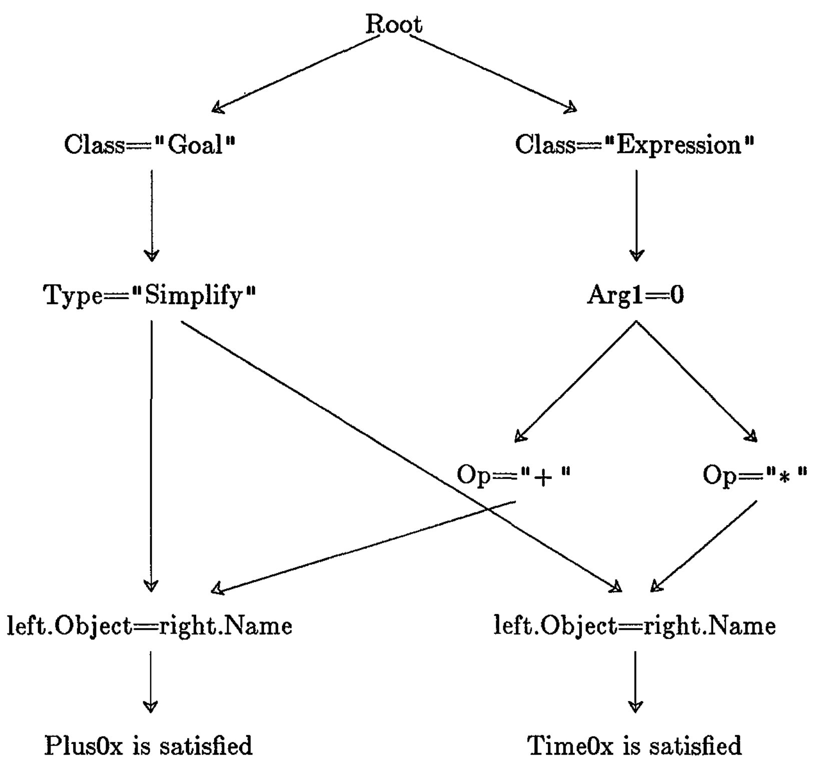

Figure 3: The Rete Network for the two rules of Example 2

3.2. DBMS Implementation of the Rete Network

Assuming secondary storage is used to store the WM elements, a straightforward implementation for the Rete Network is possible. First, all classes can be simulated by relations. The OPS5 manual provides a way to define classes of data elements using the **literalize** command. For example,

```
(literalize Emp name age salary dno)
(literalize Dept dno dname floor manager)
```

is equivalent to defining two relations `Emp` and `Dept` in a relational DBMS, except types are not explicitly defined. Using the above, the working memory can reside on secondary storage and be persistent. As with the implementation of OPS5, single input nodes need not store the tokens (or tuples), since they are simply used to filter incoming tuples. Relevant tuples are propagated to successor nodes, irrelevant ones are discarded. The only place where tokens have to be stored is two–input merge nodes. This corresponds to the case of relational joins. The approach taken for the implementation of OPS5 is to keep a log of all tokens inserted to the system so that future arrivals can be checked for matching. We will denote the two relations used to store the tokens that correspond to the left and right input of a two–input merge node by `LEFT` and `RIGHT` respectively. This leads to some redundancy, since, as in OPS5, data is stored both in the WM and the `LEFT` and `RIGHT` relations.

Before presenting the function of such a system, we give an example of a production system.

Example 3: Suppose the following two rules are defined on the `Emp` and `Dept` relations

```
        /delete Mike if he makes more than his manager/
(p R1
        (Emp   ↑ name Mike   ↑ salary <S>   ↑ manager <M>)
        (Emp   ↑ name <M>   ↑ salary {<S1> < <S>})
→
        (remove 1))

        /delete all employees working on the first floor in the Toy department/
(p R2
        (Emp    ↑ dno <D>)
        (Dept   ↑ dno <D>   ↑ dname Toy   ↑ floor 1)
→
        (remove 1))
```

For this production system, there are four relations needed, two per join condition. Let `LEFT1` and `RIGHT1` be the two relations that store tokens relevant to rule `R1` and `LEFT2` and `RIGHT2` the corresponding relations for rule `R2`. `LEFT1` will contain tuples of the form

`(Mike,<A>,<S>,<D>)`

where we have used the OPS5 notation for variables. `RIGHT1` will contain all tuples inserted in the `Emp` relation, as all of them are potential matches. Similarly, `LEFT2` will be identical to the `Emp` relation, while `RIGHT2` will contain tuples of the form

`(<D>,Toy,1,<M>)` □

We describe now the algorithm for inserting a new tuple. Deletion is handled similarly. A newly inserted tuple will be first checked for the name of the relation it belongs to ("class" checking in Figure 3) and then be propagated to the corresponding nodes of the Rete Network.

If single–input node conditions are satisfied, tuples are kept propagated until a two–input node is found. Suppose that the inserted tuple t comes from the left input of a node. Again, let LEFT and RIGHT be the relations corresponding to the left and right input respectively. Relation RIGHT is searched for tuples that match t. If at least one such tuple t_r is found, a new tuple, the join of t and t_r, is created and propagation continues. Otherwise, t is stored in LEFT and waits for a joining tuple to arrive from the right input. Tuples can never be deleted from LEFT and RIGHT relations unless there is an explicit deletion of a WM element (**remove** command).

The above method is a straightforward implementation of a Rete Network in a DBMS environment and offers several advantages, such as simplicity and re–usability of existing technology. The compilation process used in OPS5 can be used to obtain a Rete Network for a given set of rules; the above guidelines can then be used for a simple DBMS implementation. However, there are also several disadvantages.

First, conditions on the same relation (one–variable selections) may be checked at several points in the Rete Network. This is not a good tactic in a database environment where operations are set–oriented. One would like to check all conditions on a given relation with a single scan of the relation or even using indices, if they exist. Second, the Rete Network implements only one possible way of processing a set of conditions (i.e. qualifications) over a set of relations. Database technology provides more efficient ways of generating efficient access plans. Moreover, since it is the case that multiple conditions have to evaluated and these conditions may share simpler conditions, such as selections or joins, it would be advantageous to build a *global* compiled plan that avoids multiple relation accesses. Recent work on multiple–query processing [SELL86,CHAK86,PARK87] has studied this problem and algorithms can be used to generate very efficient access plans, i.e. Rete Networks. It is essential that this technology is used to improve the performance of the matching process. Finally, we find that another disadvantage of the Rete Network lies in its hierarchical structure. There is no reason why sequential propagation of tokens must be performed. For example, in the case of a three–way join condition, checking for matching tuples can be done at the same time between all three relations. Since the structure of the conditions (expressions) is known at compile time, there is no reason why one should imply a strict order in the evaluation of the one– or two–input node conditions. "Flattening" the hierarchy is another alternative, and is of significant interest in the case of a relational DBMS, where data is kept in flat tables without any structure. Such an approach is taken in Section 4.

Before moving to describe this approach, we make a note on the relationship between the above simple implementation and the one undertaken by POSTGRES [STON86b]. Notice that the Rete Network augments rule conditions with data tuples. This is done by actually storing in–coming data at the nodes of the network. POSTGRES uses a *dual* approach, i.e. it stores identifiers of possibly qualifying rules with the data. For example, in our employee database, markers are set on `Emp` tuples that possibly satisfy the conditions of rules `R1` and `R2`. In the absence of indices however, that means marking all tuples in the `Emp` and `Dept` relations. The space overhead incurred in such an implementation is clearly lower than that of the Rete Network, as rule identifiers require much less space compared to the full data tuples that the Rete Network stores. However, the process of identifying qualifying rules is more expensive in POSTGRES, as more false drops may arise. For example, in the case where all `Emp` tuples are marked because of rules `R1` and `R2`, a new insertion to that relation will trigger both of these rules, even though it should not be fired because there are no matching `Dept` tuples. POSTGRES will of course check the conditions of the rules before the corresponding actions are performed, but that will incur unnecessarily high computation cost. We discuss these trade–offs in more detail in Section 4.

4. A DBMS Approach

In the previous section we described a straightforward DBMS implementation of the Rete Network. However, working in a DBMS environment may call for several modifications to the direct implementation. First of all, the large number of intermediate relations (i.e. `LEFT` and `RIGHT` relations of each two–input node) is not realistic. Second, the performance could be poor because of the hierarchical structure of the network. For example, the propagation delay of inserting a token into C_1 (see Figure 1) will be significant if the number of single input nodes n is large. No speed–up by parallel processing is possible because all operations must be done sequentially. Flattening the hierarchy is a promising solution to the problems mentioned above. In the following we study two different approaches.

4.1. Eliminating Redundancy

The first alternative is to treat the LHS of each rule as a query to be evaluated against working memory elements, thus eliminating the need of any redundant storage. This has also been proposed in [MIRA84]. Instead of storing a large number of intermediate relations, we will only need to store one relation per class of working memory (WM) elements. Each relation records all the conditions related to that particular class of WM elements. The number of relations is thus equal to the number of classes which is relatively small compared to the number of intermediate relations used in a Rete Network. Moreover, the number of these relations is also independent of the number of the rules. We discuss the data structures and the algorithms involved in this implementation, in the following two sub-sections.

4.1.1. Data structures

There are two basic types of relations: the Working Memory Relations (WM) and the Condition Relations (COND). As discussed in Section 3.2, each class of working memory elements is stored as a WM relation. All condition elements in rules that refer to a class of WM elements, say *C*, are stored in a corresponding COND relation. Given a set of rules, the translation from a Rete Network into WM and COND relations is straightforward. For example, the rule set of Example 2 can be represented as two COND relations:

COND-Goal

Rule-ID	Type	Object
PlusOx	Simplify	<N>
TimeOx	Simplify	<N>

COND-Expression

Rule-ID	Name	Arg1	Op	Arg2
PlusOx	<N>	0	'+'	<X>
TimeOx	<N>	0	'*'	<X>

Rule-ID is the unique identifier assigned to the rule using the **p** command. Another global relation (RULE-DEF) is needed for storing the remaining information for all rules. The following table shows this relation for Example 2.

RULE-DEF

Rule-ID	Cond#	Check
PlusOx	1	0
PlusOx	2	0
...	...	..

RULE-DEF contains one tuple for each condition of each rule. Cond# shows which condition elements this tuple refers to, while the Check bit indicates whether the corresponding condition element is satisfied or not. A rule is put into the conflict set if all its Check bits are set (meaning all condition elements of the rule are satisfied.)

4.1.2. The simplified algorithm

Given the above data structures, a simple algorithm can be devised. When a working memory element (tuple) *W* of class *C* is inserted, first the tuple is inserted into the WM relation of *C* (abbreviated WM-C hereafter). Second, the COND relation of *C* (abbreviated COND-C) is searched against *W*. There are two kinds of variables used in OPS5, which are represented by symbols enclosed by <>, e.g. <N> and <X> in the example above. The first kind of variables, like <N> in Example 2, are used as a means of connecting two or more condition elements. They are the two-input nodes in the Rete Network and correspond to joins in a DBMS approach. The second kind of variables, like <X> in the same example, are just don't-care attributes. In the discussion below, "variables" means the former ones. Don't-care attributes are represented by '*' and will match anything.

For variable-free condition elements, a simple selection on COND-C is sufficient. The condition element is satisfied if *W* matches all attributes specified in the condition element. The corresponding Check bit is then set in the RULE-DEF relation. For condition elements with variables, the procedure is more complicated. A join of related WM relations is needed to determine if a specific condition is satisfied. In the simple two-way join case, however, the join degenerates into a selection on the WM relation. For example, the insertion of working memory element (Goal Simplify TERM) will cause the selection on WM relation Expression for tuples (TERM 0 '+' *) and (TERM 0 '*' *). Deletion of tuples is handled similarly. For multiple-join conditions, the system will have to come up with optimal plans for processing the queries that correspond to the LHS's of the various rules. In general, the performance of the system largely depends on the efficiency of processing joins.

In terms of space, this algorithm is much better than the Rete Network because no intermediate results are stored. On the other hand, the speed may be slower in some cases since re–computation of joins is necessary whenever a change is made to the working memory. One advantage of this alternative is that the order of joins is not fixed and can be optimized by the DBMS, compared to the fixed access plan of a Rete Network. In addition, for the case of variable–free conditions, that is single relation conditions, one can use intelligent indexing techniques such as R –trees [GUTT84] or R^+–trees [SELL87], as suggested in [STON86a], to check if a given tuple satisfies conditions stored in the COND relations.

A second alternative seeks to avoid re–computation (especially of joins) whenever changes are made to the WM. This solution also avoids storing intermediate results. In the alternative described above, the COND relation for each class of WM elements only stores information from that class. In the approach to be described next, the COND relation associated with class C also stores information from other classes that interact with C in productions. In other words, instead of propagating changes and storing intermediate results, as the Rete Network does, this alternative propagates changes and stores that information in the COND relations of the affected classes. This approach is detailed in the next sub–section.

4.2. The New Approach

The main design goal of our approach is to speed up the matching process. The simplified algorithm described above suffers from the fact that joins have to be re–computed every time. To tackle that problem, we introduce the idea of *matching patterns* which alleviates the problem of recomputation. As above, we first describe the data structures used and then the algorithms for handling insertions and deletions of tuples.

4.2.1. Data structures: Trading space for time

Our approach also uses condition (COND) relations. Variable free conditions are handled in exactly the same way as in the simplified algorithm, and therefore are omitted in the discussion below. Each tuple in the COND relation has the following attributes:

(1) Rule ID (RID), to record the unique rule identifier

(2) Condition Element Number (CEN), to differentiate among conditions of the same rule

(3) Restrictions on each attribute of the corresponding WM relation

(4) A list of Related Condition Elements (RCE), each RCE being represented by a (RID,CEN) pair (see later discussion)

(5) A Marker, comprised by one bit per RCE, default to zero.

The structure is best described by an example:

Example 4: Assume three relations A, B, C, with attributes Ai, Bi, and Ci, i=1,2,3 respectively. The following is a rule definition.

```
(p Rule-1
     (A   ↑ A1 <x>   ↑ A2 'a'  ↑ A3 <z>)
     (B   ↑ B1 <x>   ↑ B2 <y>  ↑ B3 'b')
     (C   ↑ C1 'c'   ↑ C2 <y>  ↑ C3 <z>)
→
     ( ... ))
```

Rule-1 has three conditions (three-way join) involving relations A, B and C respectively. We have three COND relations: COND-A, COND-B, and COND-C. These three relations are related to each other by variables <x>, <y> and <z> because of the conditions of Rule-1. The initial contents of these COND relations are as follows:

COND-A						
RID	CEN	A1	A2	A3	RCE	Mark: BC
Rule-1	1	<x>	'a'	<z>	(B,2),(C,3)	00

COND-B						
RID	CEN	B1	B2	B3	RCE	Mark: AC
Rule-1	2	<x>	<y>	'b'	(A,1),(C,3)	00

COND-C						
RID	CEN	C1	C2	C3	RCE	Mark: AB
Rule-1	3	'c'	<y>	<z>	(A,1),(B,2)	00

When a WM element is inserted into relation A, two tasks are executed. First, we have to examine the tuples in COND-A and determine if this element satisfies Rule-1 (as well as any other rule that is defined on A). The second task is to do the equivalent of propagating changes through the Rete Network and store information on how class A elements interact with class B and C elements within Rule-1. This information is stored in the form of "matching patterns" (see discussion that follows) in COND-B and COND-C. □

The Related Condition Elements list is used to show which conditions of the same rule are affected because of insertions or deletions in the relation examined. There is one Mark bit for each RCE, which if set indicates that the "matching pattern" is created by the corresponding condition element. A tuple in a COND relation with at least one Mark bit set is called a *matching pattern.* The matching pattern is the key point of the whole algorithm which improves the matching process. A matching pattern in a COND relation indicates that there is some tuple in another (related) WM relation having the property of the matching pattern and therefore is joinable with tuples in the current WM relation. Hence, when a tuple is inserted later in the current WM relation which matches the matching pattern, we know immediately that there is a match. The details of the algorithm are discussed next.

4.2.2. The algorithm

When a working memory element of class C is inserted, say tuple t, the system performs the following:

```
Search relation COND-$C$ for tuples matching $t$.
**For** each matching tuple $T$, **do**
**begin**
    If $T$ contains a variable-free condition, or all Mark bits of $T$ are set, set the
    Check bit(s) in the RULE-DEF relation.
    **For** each RCE($X$,$n$) of $T$ **do**
```

```
        begin
            Search relation COND-$X$ for tuples $M$ matching the pattern desired
            (which can be derived from the definition of the rule) with the restriction
            that the RID must be the same as that of $T$ and the CEN is equal to $n$.
            Furthermore, each Mark bit must be set in $T$ if the corresponding Mark
            bit is set in the matching tuple $M$.
            For each tuple $M$ found as described, do
            begin
                Unify $M$ with the desired pattern. If a new binding is introduced,
                create a new tuple with the new binding and set the Mark bit of $C$.
            end
        end
    end
```

Let us trace the algorithm for the rule of Example 4.

Example 5: Suppose that we insert the tuples B(4,5,b), C(c,7,8), A(4,a,8) and B(4,7,b) in the sequence given. The contents of the various COND relations will be as follows:

COND-A							
RID	CEN	A1	A2	A3	RCE	Mark: BC	Comment
Rule-1	1	<x>	'a'	<z>	(B,2),(C,3)	00	Original Tuple
Rule-1	1	4	'a'	<z>	(B,2),(C,3)	10	By tuple B(4,5,b)
Rule-1	1	<x>	'a'	8	(B,2),(C,3)	01	By tuple C(c,7,8)
Rule-1	1	4	'a'	8	(B,2),(C,3)	11	By tuple B(4,7,b)

COND-B							
RID	CEN	B1	B2	B3	RCE	Mark: AC	Comment
Rule-1	2	<x>	<y>	'b'	(A,1),(C,3)	00	Original Tuple
Rule-1	2	<x>	7	'b'	(A,1),(C,3)	01	By tuple C(c,7,8)
Rule-1	2	4	<y>	'b'	(A,1),(C,3)	10	By tuple A(4,a,8)
Rule-1	2	4	7	'b'	(A,1),(C,3)	11	By tuple A(4,a,8)

COND-C

RID	CEN	C1	C2	C3	RCE	Mark: AB	Comment
Rule-1	3	'c'	<y>	<z>	(A,1),(B,2)	00	Original Tuple
Rule-1	3	'c'	5	<z>	(A,1),(B,2)	01	By tuple B(4,5,b)
Rule-1	3	'c'	<y>	8	(A,1),(B,2)	10	By tuple A(4,a,8)
Rule-1	3	'c'	5	8	(A,1),(B,2)	11	By tuple A(4,a,8)
Rule-1	3	'c'	7	<z>	(A,1),(B,2)	01	By tuple B(4,7,b)
Rule-1	3	'c'	7	8	(A,1),(B,2)	11	By tuple B(4,7,b)

Notice that when B(4,7,b) is inserted, the last tuple in COND-B causes Rule-1 to be put in the conflict set because all Mark bits are set. □

The deletion of a working memory element could remove rules from the conflict set. The basic algorithm is very similar to the insertion algorithm discussed above. The difference is that instead of setting Mark bits, we reset them in the case of a match and instead of inserting new matching patterns, we delete existing ones. However, because a matching pattern tuple may have been created by more than one WM element, deleting one of them is not enough to delete the matching pattern tuple. To handle such cases, Mark bits can be easily replaced by counters to record the number of contributing tuples. The propagation algorithm is modified by replacing setting/resetting of Mark bits with incrementing/decrementing the corresponding counters.

Another issue not discussed in previous sections is the capability that OPS5 offers in defining negated conditions. Condition elements preceded by the '–' operator are interpreted as negated conditions. Such a condition will be satisfied if there is no WM element satisfying the corresponding non–negated condition element. To incorporate negated condition elements in our algorithm, some modifications need be made. First of all, the default values for the Mark bits of such a condition element should be set to one instead of zero. Second, the insertion of a new tuple that matches the condition will have to reset the corresponding Mark bit, instead of setting it. Similarly, deletion of a working memory element will do the opposite. Hence, negated conditions can be supported easily.

4.2.3. Discussion

There are several parameters that we can use to compare our approach to the previously mentioned alternatives.

Time

Matching is very fast with our approach because only a single search over a COND relation is necessary. Maintenance of matching patterns is more expensive since propagation of matching patterns must be performed. The propagation cost though, is the same as the cost incurred by a Rete Network. However, our approach is easily parallelizable, since propagation of changes can be performed in parallel to all the COND relations. In contrast to that, the Rete Network method is highly sequential. More important, in our approach, the conflict set is updated first, and then the maintenance process follows. In the Rete algorithm, propagation through the discrimination network must precede the updates to the conflict set; rule execution is thus delayed further.

Space

Clearly, our approach consumes a lot of space for storing matching patterns. As mentioned above, this is a trade-off between matching time and space. Notice that the matching patterns are actually the result of joins we have so far computed plus other associated information. Therefore, we are actually doing the join in an incremental way, thus reducing processing time. Compared to the Rete Network, the results of joins are stored in a better form (COND relations), so that matching is reduced to a search and can be done efficiently.

There are several possible improvements to our approach. First, it is obvious that there is a lot of redundancy among matching patterns. Compacting them in a nice way without sacrificing performance is crucial in applications with limited space. We are currently exploring several compaction techniques. Second, since a lot of selection operations are used in the algorithm, efficient implementation of selection, i.e. variable–free condition checking, is very important. Building indices such as R–trees or R^+–trees on COND relations can help in speeding up this process. Another significant advantage of such indices is their use in answering queries on the rulebase itself. For example, questions of the form

Give me all the rules that apply on employees older than 55 can be easily answered using such an index. Supporting rulebase queries is very important in the design of expert database systems that can provide information on the effect of various rules, even if data that satisfy the conditions of the rules *has not* already been stored in the database [LIN87]. Notice that this is not possible in systems, such as POSTGRES, where rule information is stored together with the actual data. Finally, as discussed also above, our scheme can be fully parallelized. Parallel processing can help improving the propagation and maintenance of matching patterns.

After discussing the matching and maintenance processes of our approach, we move next to present our ideas on the second major part of a rule based system, namely the execution of applicable rules.

5. Processing Applicable Rules

The RHS actions of productions or rules placed in the conflict set must be executed. The actions on the RHS of the production represent changes to the WM classes and include insertions, deletions and updates of WM elements. In our DBMS implementation, executing these actions means that these changes must be made to the corresponding WM relations. The changes will trigger the maintenance process, described above, that affects the COND relations. RHS actions that add or delete tuples from WM relations trigger the insertion or deletion algorithm, respectively. An update is equivalent to a delete followed by an insert, and trigger both algorithms. Conceptually, execution of a production completes after processing changes against both WM relations and the COND relations.

5.1. Sequential versus Parallel Execution

In the Rete network implementation of OPS5, productions placed in the conflict set are executed in a serial order. In each cycle, a single production, together with the corresponding tokens satisfying it, is selected. The RHS actions are then executed; this may result in changes to WM. In the next match phase, these updates to WM are propagated through the discrimination net. Consequently, productions in the conflict set may be deleted, or productions may be added.

In our proposed DBMS implementation of a production system, changes to the WM relations will be propagated to the COND relations and matching patterns will be added to the COND relations. When a tuple that is inserted into a WM relation matches a tuple in the corresponding COND relation with all its bits set, then, the matching pattern will be added to the conflict set; this is equivalent to adding a production. The matching pattern tuple, however, does not store pointers to, or identifiers of the actual tuples of the WM relations. These tuples must be selected before executing the RHS actions. The attribute values in each matching pattern will provide the selection criterion that must be applied when selecting tuples from the WM relations.

When several combinations of WM elements satisfy a single production, the Rete implementation treats each combination as a separate instance of the production, and each instance is executed independently. Traditionally, DBMS support set–at–a–time processing, against all tuples of a relation. Thus, in our proposed implementation, sets of tuples from each WM relation, satisfying the selection criterion, will be grouped together. A selected production will execute simultaneously against all combinations of these sets of tuples.

Each production in the conflict set (together with all tokens in the Rete implementation) or each matching pattern (with its combinations of tuples from the WM relations in the DBMS implementation) can be treated as a transaction that is to be executed. The Rete implementation will then resemble a serial execution strategy of transactions in the conflict set. We are interested in exploring a concurrent execution strategy, for the DBMS implementation.

We use serializability to determine if the concurrent execution is equivalent to a particular serial execution strategy. We assume that a locking strategy will be used to ensure the execution is serializable. We show that serializability requires that appropriate locks be placed on both the WM and COND relations.

5.2. Equivalence of a Serial and Parallel Execution Strategy

We examine how the serializability criterion can be used to show the equivalence of a serial and concurrent (interleaved) execution strategy, in a production system environment. We also use serializability to show that the concurrent execution of a set of productions in the conflict set maintains the consistency of the database.

Given an initial set Ψ_1 of transactions, each of which corresponds to an already satisfied production in the conflict set, we compare the serial execution of these transactions in a serial production system environment (OPS5) with their interleaved execution in a concurrent environment (our proposal).

In a serial production system, in each step i, a single transaction, T_i, is arbitrarily selected from the conflict set and applied (Select and Act operations of Section 2.1). Subsequently, the production system will determine (Match operation of Section 2.1) if, as a result of applying T_i, some other transactions in this conflict set are no longer applicable; if so, these transactions will be deleted from the set. Let the set of transactions deleted in step i be Δdel_i. Also as a result of applying T_i, the production system will determine (Match cycle) if some additional transactions are now applicable, as well. Let the set of transactions added in step i be Δadd_i. The new set of candidate transactions in step $i+1$ is $\Psi_{i+1} = \left\{\Psi_i - T_i - \Delta del_i + \Delta add_i\right\}$. This process will continue until finally in step F, the set Ψ_F is empty. Note that in the serial production system, each step corresponds to the execution of a single transaction.

The selection of each T_i is arbitrary; thus, it is entirely possible that in step 2, T_2 is selected from the set $\left\{\Psi_1 - T_1 - \Delta del_1\right\}$ which is the set $\left\{\Psi_2 - \Delta add_1\right\}$. In other words T_2 could also be selected from the initial set Ψ_1 and not from the added set of transactions Δadd_1. Similarly, in subsequent steps i, T_i can be selected from the set $\left\{\Psi_1 - \sum_{j=1}^{i-1}\left(T_j - \Delta del_j\right)\right\}$ which is the same as the set $\left\{\Psi_i - \sum_{j=1}^{i-1}\left(\Delta add_j\right)\right\}$.

If the selection is as described, then, eventually after some f_1 steps, the serial production system will have executed a sequence of f_1 transactions T_1, T_2, .. T_{f_1}, where each T_i happens to be an element of the initial set Ψ_1. After step f_1, all transactions in Ψ_1 are either executed or deleted and the set of applicable transactions for step (f_1+1), Ψ_{f_1+1} is the set $\left\{\sum_{j=1}^{f_1} \Delta add_j\right\}$, i.e., all the transactions added in the f_1 previous steps, which were not selected previously. In step (f_1+1), T_{f_1+1} is chosen from this set Ψ_{f_1+1}.

We have gone through this exercise with the serial production system to compare it with a concurrent execution strategy. Given an initial set of transactions, Ψ_1, the serial production system can arbitrarily select and execute an initial sequence of transactions all of which happen to be in Ψ_1. Given this same initial set Ψ_1, a concurrent execution strategy would interleave the execution of this set of transactions. If an appropriate protocol is used, and the resulting schedule is serializable, then it must be equivalent to some serial schedule T_1, T_2, .., etc., where each T_i must be from the initial set Ψ_1. In other words, the concurrent production system will execute an equivalent serial schedule which may be the same as the serial schedule arbitrarily selected by the serial production system.

After the interleaved execution of the set Ψ_1 completes, the equivalent serial schedule of the concurrent production system is the same as the (arbitrary) serial schedule of the serial production system, and a sequence of f_1 transactions would have been executed. The second conflict set will be identical to the set Ψ_{f_1+1} which was available to the serial production system in step (f_1+1). Thus, the serial execution and the concurrent execution are found to be equivalent.

We have now to examine this equivalence in the environment of the WM and COND relations. We must show that the conditions for serializability are enforced, or that locks on the WM and COND relations are obtained at the appropriate time, and more importantly not released until some logical *commit* point for the production is reached. We have to define what that point is.

Selecting an appropriate commit point must satisfy two requirements in the concurrent production system environment. First, the interleaved execution of a set of productions must maintain consistency of the database. i.e., two transactions that update the same WM relation must be serializable. Second, transactions that are inter–related and effect each others execution, i.e., transactions that delete each other's matching pattern tuples from the conflict set, must interact correctly. For example, when a transaction T_i executes, the selected commit point must enforce a delay in the execution (and commit) of the transactions in the set $\sum_{j=1}^{i-1} \Delta del_j$, so that their subsequent execution can be prevented or aborted.

For our DBMS implementation, this logical commit point must necessarily occur after the maintenance process (insertion and deletion algorithms triggered by changes to the WM relation) finishes updating the COND relations. The following description will justify this necessity.

We say that a transaction (production) is positively dependent on a WM relation if the LHS of the production is satisfied by the existence of some specific tuples of a WM relation. A transaction is negatively dependent on a WM relation if it is satisfied by the absence of some specific tuples. A transaction is independent of a WM relation if it is unaffected by the existence or absence of specific tuples.

In the current definition of the OPS5 language, the RHS actions of a production can only delete or update a WM element if its existence is tested on the LHS; i.e., a transaction can only delete tuples from working memory relations on which it is positively dependent. However, a transaction can insert tuples into any WM relation.

A transaction, corresponding to a selected production, first retrieves the matching pattern tuple from the conflict set. This pattern does not store tuple identifiers for the tuples of WM relations satisfying the production. Attribute values from the matching pattern tuple are used to generate selection predicates for the affected WM relations. A read lock must be placed on those WM relation tuples that are retrieved.

A transaction T_i that is positively dependent on a WM relation R_i may delete or update specific tuples of R_i. This may affect the execution of another transaction T_j that is also positively dependent on R_i. Both T_i and T_j must initially obtain a read lock on the specific tuples of R_i. If T_j completes execution first, it will release its read lock, and T_i may proceed to delete or update R_i. If T_i requests a write lock on the same tuples that are read locked by T_j, then the execution of T_i will be delayed until T_j completes execution. In both cases, T_j precedes T_i in the equivalent serial execution schedule, and the database is consistent.

If T_i obtains a write lock (and thus, completes execution) before T_j requests a read lock, then T_j may be in the set Δdel_i, so its execution must be delayed until after the update or delete from R_i. Changes made to R_i trigger the maintenance process and propagate changes to the COND relations. The maintenance process can potentially delete the matching pattern tuple for T_j from the conflict set. For this reason, T_i must not commit and release its locks until the maintenance process completes. If the matching pattern tuple corresponding to T_j is unaffected or is deleted before T_j starts execution, then, no further action is required. If T_j has already started execution, it will be delayed since it will not be able to obtain a read lock until T_i

releases its write lock on the tuples of R_i. Now, even if the matching pattern tuple for T_j is deleted, T_j will still be executed. However, T_j will not be able to process tuples of R_i that have already been deleted by T_i so the database will still be consistent. It is also possible that both T_i and T_j delete or update tuples from R_i, and that T_i is in the set $\Delta\, del_j$ and vice versa. This could lead to a deadlock of the two transactions.

A transaction T_i that is negatively dependent on a WM relation R_i can be affected by another transaction T_j that inserts specific tuples into R_i. T_i may be in the set $\Delta\, del_j$ and its execution must be delayed, if T_j executes first. T_j will always need a write lock on R_i before it can be executed. If T_i is required to obtain a read lock on the entire relation R_i, and if it obtains the read lock on R_i first, then the execution of T_j will be delayed. T_i will precede T_j in the serial execution schedule, and the database will be consistent.

If T_j obtains a write lock first, then T_i will be delayed. Now, T_j must not commit and release this lock until the insertion into R_i and the maintenance process triggered by the insertion is complete. This is because the matching pattern tuple for T_i may be deleted from the conflict set, during the maintenance process. If the matching pattern tuple for T_i is unaffected or if it is deleted before T_i starts execution, then, no further action is required.

If T_i has already started execution, then its execution will only be delayed, but it will not be prevented even if the corresponding matching pattern tuple is deleted. Since T_i does not lock specific tuples of R_i but locks the entire relation, executing T_i may make the database inconsistent. Thus, if the corresponding matching pattern tuple has been deleted, then T_i must not be allowed to commit, and the system must abort T_i. A better solution would require that the DBMS support the NOT EXISTS operator through its querying facility. Now, a transaction T_i that is negatively dependent on R_i will have to obtain a read lock on the entire R_i relation, and verify that the specific tuples of R_i do not exist, before being allowed to execute its RHS actions. This will ensure serializability.

To summarize, we have shown that the serial execution of productions in the Rete implementation and the concurrent execution in the proposed DBMS implementation are equivalent. We assumed a locking strategy was used to enforce serializability and justified our claim that a production should not commit its RHS actions (changes to WM relations) and release its locks on these WM relations, until the triggered maintenance process updates the affected COND relations as well.

The benefits of concurrent execution can be measured in several ways. First, the number of operations that must execute in a non–interleaved fashion measures the time of execution. In the best case, neglecting locking overhead, this will be proportional to the maximum number of updates to any WM relation or COND relation. In the worst case, this will reduce to the time taken for a serial execution. A second measure that is proposed is the number of serializable schedules equivalent to a single serial schedule. This measure is proportional to the number of possible choices of actions that can be executed at any instant. Details of these estimates are in [RASC87].

6. Conclusions

We have studied the problem of storing, maintaining and using large production rule bases. Starting with the known Rete Network method, we provided first an easy implementation of the OPS5 data structures and matching algorithm for a DBMS environment, and then suggested a new approach with similar space requirements but offering the advantage of faster detection of applicable rules. As the problem of maintaining a set of condition–action rules is the same as the problem of maintaining materialized views and triggers, our method can be used for these latter problems as well.

The approach we have taken achieves localization of the match procedure in the sense that a single relation has to be checked in order to decide if an inserted or deleted tuple renders a rule applicable for firing. This feature not only is suitable for relational DBMS's but in addition makes our method easily parallelizable. Operations on the database can be processed in parallel since we have managed to eliminate the hierarchical propagation delay associated with the Rete Network.

Finally, we have proposed a new way to process applicable rules based on the notion of transactions. Again, the goal was to eliminate the sequential nature of the OPS5 system. Applicable rules can be processed concurrently, assuming that the DBMS will serialize RHS actions (insertions and deletions) through its concurrency control mechanism.

Our current work focuses on the details and the optimization of the proposed approach. First, we examine the ways in which multiple query processing and optimization algorithms can be applied to provide optimal Rete Networks. Although our approach does not assume any global execution strategy, we are interested to conduct a performance analysis of the original Rete Network, a Rete Network which has been optimized using multiple–query processing heuristics [SELL88] and our approach. Second, we look into the details and the extensions needed to R^+–trees in order to use them as fast matching devices on COND relations [LIN87]. Finally, we study the properties and performance of a fully concurrent system where transactions are used to implement the actions of the various rules. In particular, we look at concurrency control methods based on locking. Estimates for assessing the benefits of concurrent execution are being investigated [RASC87].

7. References

[BANC86] Bancillhon, F., and Ramakrishnan, R. An Amateur's Introduction to Recursive Query Processing. *Proceedings of the ACM–SIGMOD International Conference on the Management of Data,* Washington, DC (1986).

[BLAK86a] Blakeley, J.A., Larson, P., and Tompa, F.W. Efficiently Updating Materialized Views. *Proceedings of the ACM–SIGMOD International Conference on the Management of Data,* Washington, DC (1986).

[BLAK86b] Blakeley, J.A., Coburn, N., and Larson, P. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *Proceedings of the 12th International Conference on Very Large Data Bases,* Kyoto, Japan (1986).

[BUNE79] Buneman, O.P., and Clemons, E.K. Efficiently Monitoring Relational Databases. *ACM Transactions on Database Systems* (4) 3 (1979).

[CHAK86] Chakravarthy, U.S., and Minker, J. Multiple Query Processing in Deductive Databases. *Proceedings of the 12th International Conference on Very Large Data Bases,* Kyoto, Japan (1986).

[ESWA76] Eswaran, K.P. et al. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* (19) 11 (1976).

[FORG81] Forgy, C.L. OPS5 User's Manual. Technical Report CMU–CS–81–135, Carnegie–Mellon University (1981).

[FORG82] Forgy, C.L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence* (19) (1982).

[GRAY78] Gray, J.N. Notes on Data Base Operating Systems. IBM Research, Technical Report RJ–2254 (1978).

[GUTT84] Guttman, A. R–Trees: A Dynamic Index Structure for Spatial Searching. *Proceedings of the ACM–SIGMOD International Conference on the Management of Data* (1984).

[HANS87] Hanson, E.N. Efficient Support for Rules and Derived Objects in Relational Database Systems. Ph.D. Thesis, Computer Science Division, University of California, Berkeley (1987).

[HAYE85] Hayes–Roth, F. Rule Based Systems. *Communications of the ACM* (28) 9 (1985).

[IOAN86] Ioannidis, Y. Processing Recursion in Deductive Database Systems. Ph.D. Thesis, Computer Science Division, University of California, Berkeley (1986).

[KERS86] Kershberg, L., Editor. *Expert Database Systems: Proceedings From the First International Workshop.* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1986).

[KERS87] Kershberg, L., Editor. *Expert Database Systems: Proceedings From the First International Conference.* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA (1987).

[LIN87] Lin, C–C., and Sellis, T. Using Geometric Structures to Manage Constraints in Large Knowledge Bases, Department of Computer Science, University of Maryland, College Park (unpublished manuscript).

[MIRA84] Miranker, D. Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE*. *Proceedings of the International Conference on First Generation Computer Systems,* Japan (1984).

[PARK87] Park, J., and Segev, A. Using Common Subexpressions to Optimize Multiple Queries. Lawrence Berkeley Laboratory Technical Report LBL–23597, University of California, Berkeley (1987).

[RASC87] Raschid, L. The Design and Implementation Techniques for an Integrated Knowledge Base Management System. Ph.D. Thesis, Department of Electrical Engineering, University of Florida, Gainesville (1987).

[SELL86] Sellis, T. Global Query Optimization. *Proceedings of the ACM–SIGMOD International Conference on the Management of Data,* Washington, DC (1986).

[SELL87] Sellis, T., Roussopoulos, N., and Faloutsos, C. The R^+–tree: A Dynamic Index for Multi–Dimensional Objects. *Proceedings of the 13th International Conference on Very Large Data Bases,* Brighton, England (1987).

[SELL88] Multiple–Query Optimization. *ACM Transactions on Database Systems* (1988), to appear.

[STON75] Stonebraker, M., Implementation of Integrity Constraints and Views by Query Modification. *Proceedings of the ACM–SIGMOD International Conference on the Management of Data,* San Jose, CA (1975).

[STON76] Stonebraker, M., Wong, E., Kreps, P, and Held, G. The Design and Implementation of INGRES. *ACM Transactions on Database Systems* (1) 3 (1976).

[STON85] Stonebraker, M. Triggers and Inference in Data Base Systems. *Proceedings of the Islamorada Expert Database Systems Conference* (1985).

[STON86a] Stonebraker, M., Sellis, T., and Hanson, E. Rule Indexing Implementations in Database Systems. In [KERS87].

[STON86b] Stonebraker, M., and Rowe, L. The Design of POSTGRES. *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, Washington, DC (1986).