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

Timos Sellis[1], **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. In this paper we study methods for storing and manipulating large rule bases using relational database management systems. First, we provide a matching algorithm 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. The proposed method is fully parallelizable, which makes its use even more attractive, as it can be used in parallel computing environments.

This research was sponsored partially by the National Science Foundation under Grant CDR-85-00108 and by UMIACS.

[1] 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 mentioned 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.

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. In the case of multiple and recursive definitions new execution mechanisms need to be incorporated [1,16]. More general kinds of rule systems, such as production rule systems [11], 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 briefly 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. The reader is referred to [17] for a more detailed presentation.

2. Production Rule Systems

Both the DBMS and AI community have been studying problems related to production rules, usually under different contexts. Hanson [10] 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* 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 priorities or, in general, order rules according to some static or dynamic criteria and then fire the rules in that order. 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.

2.2. The AI Way

The most representative approach to efficient matching has been the Rete Match Algorithm [8] used in the OPS5 system [7]. 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. □

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 [6] in the context of concurrency control and authorization. Perhaps

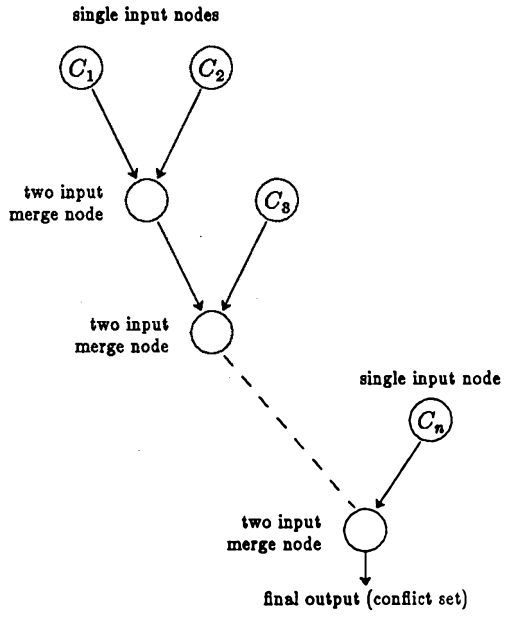

Figure 1: Example of a Discrimination Network

the first systematic work has been the one by Buneman and Clemons in [4] where they examine 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. 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). Similar work has been done on the same subject can be found in [2,3].

Recently, Stonebraker proposed an extension to QUEL commands to model triggers [18]. 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*). In [19], mechanisms to efficiently detect qualifying rules in a DBMS environment have been suggested (*rule indexing*).

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 [8]. 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

```
(p R1
  (Emp ↑Name Mike ↑Dno <D>)
  (Dept ↑Dno <D> ↑D Toy ↑Floor 1 ↑Mgr <M>)
→  (remove 1))

(p R2
  (Emp ↑Name Mike ↑Dno <D>)
  (Dept ↑Dno <D> ↑D Shoe ↑Floor 1 ↑Mgr <M>)
→  (remove 1))
```

that remove Mike from the Emp class if he works on the first floor and in the Toy department (R1) or the Shoe department (R2). The ↑ symbol is used to indicate attribute names. □

The Rete Match Algorithm is used in OPS5 to reduce the computation required to check for conditions that are satisfied. The conditions of the various rules are evaluated and changes in the database are monitored in an efficient way. This is achieved by keeping all matches among working memory elements, that is, all tuples satisfying selections or pairs of tuples satisfying joins. The descriptions of working memory (WM) changes that are propagated to the Rete Network are called *tokens*. The algorithm maintains also a *conflict set* which contains information on all applicable rules and the data elements that cause these rules to fire.

Rule definitions are compiled and the discrimination network is produced. Figure 2 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 of the form `attr. op constant`, where $op \in \{<, >, \leq, \geq, =, \neq\}$. Finally, *two-input* nodes are used to check joins of the form

`left.attribute` *op* `right.attribute`

A tuple t is first checked at one-input nodes to determine if it is an Emp or a Dept 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.

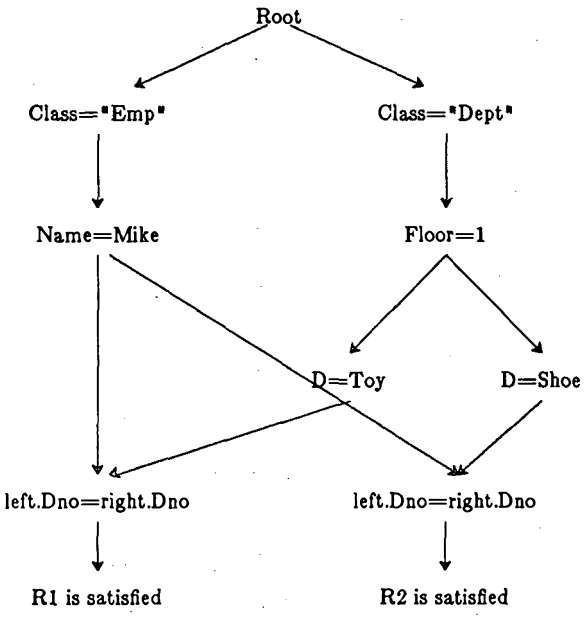

Figure 2: The Rete Network for Example 2

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 are simulated by relations. In our example, there are two relations, `Emp` and `Dept`. 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. 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. In Example 2, 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`. Assuming that the schema for the database is `Emp(Name,Salary,Dno)` and `Dept(Dno,D,Floor,Mgr)`, `LEFT1` and `LEFT2` will contain tuples of the form `(Mike,<S>,<D>)`, where we have used the OPS5 notation for variables. `RIGHT1` will contain tuples of the form `(<D>,Toy,1,<M>)`, while `RIGHT2` will contain tuples of the form `(<D>,Shoe,1,<M>)`. The algorithm for inserting a new tuple is the same as the Rete algorithm of section 3.1.

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, 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 [5,13,16]. Second, 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. "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.

4. A DBMS Approach

Working in a DBMS environment may call for several modifications to the direct implementation of the previous section. 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 [12]. 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. For example, the rule set of Example 2 can be represented as two COND relations:

COND-Emp

Rule-ID	Name	Salary	Dno
R1	Mike	<S>	<D>
R2	Mike	<S>	<D>

COND-Dept

Rule-ID	Dno	D	Floor	Mgr
R1	<D>	Toy	1	<M>
R2	<D>	Shoe	1	<M>

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. Tuples in this relation have the form (Rule-ID,Cond#,Check), RULE-DEF contains one tuple for each condition of each rule. Cond# shows which condition element 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 tuple W of class C is inserted, first the tuple is inserted into the WM relation of C. 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. <D> and <M> in the example above. The first kind of variables, like <D> 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 the DBMS approach. The second kind of variables, like <M> 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 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 tuple (Emp Mike D1) causes the selection on relation Dept for tuples (D1 Toy 1 *) and (D1 Shoe 1 *). 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 [9] or R^+–trees [15], as suggested in [19], to check if a given tuple satisfies conditions stored in the COND relations.

A second alternative seeks to avoid re–computation of joins by propagating changes and storing them 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. 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. Trading space for time

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) 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.

We illustrate the use of these attributes through the following example.

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

```
(p R3
  (A  ↑A1 <x>  ↑A2 'a'  ↑A3 <z>)
  (B  ↑B1 <x>  ↑B2 <y>  ↑B3 'b')
  (C  ↑C1 'c'  ↑C2 <y>  ↑C3 <z>)
→   ( ... ))
```

R3 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 R3. These variables, as mentioned above, are used to implement joins among the three relations. The initial contents of these COND relations are as follows (in the following we omit the field RID as it always contain the entry R3 in this example):

COND-A

CEN	A1	A2	A3	RCE	Mark BC
1	<x>	'a'	<z>	(B,2),(C,3)	00

COND-B

CEN	B1	B2	B3	RCE	Mark AC
2	<x>	<y>	'b'	(A,1),(C,3)	00

COND-C

CEN	C1	C2	C3	RCE	Mark AB
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 R3 (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 R3. This information is stored in the form of "matching patterns" (see discussion that follows) in COND-B and COND-C. □

The RCE 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*. A matching pattern 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 that 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 tuple with that binding and set the Mark bit of C.
end
end
end

We trace the algorithm for the rule of Example 3.

Example 4: 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 COND relations are as follows (again, we omit the field RID as it always contain the entry R3 in this example):

COND-A

CEN	A1	A2	A3	RCE	Mark BC
1	<x>	'a'	<z>	(B,2),(C,3)	00
1	4	'a'	<z>	(B,2),(C,3)	10
1	<x>	'a'	8	(B,2),(C,3)	01
1	4	'a'	8	(B,2),(C,3)	11

COND-B

CEN	B1	B2	B3	RCE	Mark AC
2	<x>	<y>	'b'	(A,1),(C,3)	00
2	<x>	7	'b'	(A,1),(C,3)	01
2	4	<y>	'b'	(A,1),(C,3)	10
2	4	7	'b'	(A,1),(C,3)	11

COND-C

CEN	C1	C2	C3	RCE	Mark AB
3	'c'	<y>	<z>	(A,1),(B,2)	00
3	'c'	5	<z>	(A,1),(B,2)	01
3	'c'	<y>	8	(A,1),(B,2)	10
3	'c'	5	8	(A,1),(B,2)	11
3	'c'	7	<z>	(A,1),(B,2)	01
3	'c'	7	8	(A,1),(B,2)	11

Tuples with a 00 entry in the Mark column correspond to original condition tuples. Notice that when B(4,7,b) is inserted, the last tuple in COND-B causes R3 to be put in the conflict set because all Mark bits are set. □

Due to space limitations the algorithms for supporting deletions as well as negated conditions in rules is omitted. The basic algorithm is very similar to the insertion algorithm discussed above; for more details the reader is referred to [17].

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. The propagation cost, is the same as the cost incurred by a Rete Network. 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 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 this proposal. 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. 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.

Finally, we comment on the relationship of our approach to the one used in POSTGRES [20]. 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 and Dept tuples that possibly satisfy the conditions of rules R1 and R2. 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.

5. Processing Applicable Rules

Productions placed in the conflict set must be executed. In the Rete network implementation of OPS5, in each cycle, a single production and its corresponding tokens is selected. Applying the RHS actions changes the WM elements and eventually updates the conflict set.

In our implementation, matching pattern tuples representing satisfied productions do not include identifiers to corresponding WM elements. The attribute values of each matching pattern provides a selection criterion to select the corresponding tuples from the WM relations. Changes made to WM relations will trigger the maintenance algorithm to update the COND relations and generate matching patterns for satisfied productions.

Clearly, each production and its combinations of tuples from the WM relations, can be treated as a transaction. The Rete implementation is a serial execution strategy of these transactions. In [17] we explore a concurrent execution strategy, for our DBMS implementation. We use serializability to show that the proposed concurrent execution is equivalent to a particular serial execution strategy (in the Rete implementation). Assuming a basic locking strategy, serializability requires that appropriate locks be placed on both the WM and COND relations and not be released until some *logical commit point* is reached. The interested reader is referred to [17] for further details.

There are several benefits to concurrent execution. 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 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 [14].

6. Conclusions

We have studied the problem of storing, maintaining and using large production rule bases. 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.

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 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. 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 [14].

7. References

[1] Bancillhon, F. and Ramakrishnan, R. An Amateur's Introduction to Recursive Query Processing. *Proc. of the ACM-SIGMOD Conf.*, Washington, DC (1986).

[2] Blakeley, J.A., et al. Efficiently Updating Materialized Views. *Proc. of the ACM-SIGMOD Conf.*, Washington, DC (1986).

[3] Blakeley, J.A., et al. Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates. *Proc. of the 12th VLDB Conf.*, Japan (1986).

[4] Buneman, O.P. and Clemons, E.K. Efficiently Monitoring Relational Databases. *ACM TODS* (4) 3 (1979).

[5] Chakravarthy, U.S. and Minker, J. Multiple Query Processing in Deductive Databases. *Proc. of the 12th VLDB Conf.*, Japan (1986).

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

[7] Forgy, C.L. OPS5 User's Manual. Tech. Report CMU-CS-81-135, Dept. of Computer Science, Carnegie-Mellon University (1981).

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

[9] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. *Proc. of the ACM-SIGMOD Conf.* (1984).

[10] 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).

[11] Hayes-Roth, F. Rule Based Systems. *CACM* (28) 9 (1985).

[12] Miranker, D. Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE*. *Proc. of the Conf. on Fifth Generation Computer Systems*, Japan (1984).

[13] Park, J. and Segev, A. Using Common Subexpressions to Optimize Multiple Queries. *Proc. of the 4th Data Engineering Conf.*, Los Angeles, CA (1988).

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

[15] Sellis, T., et al. The R^+-tree: A Dynamic Index for Multi-Dimensional Objects. *Proc. of the 13th VLDB Conf.*, England (1987).

[16] Sellis, T. Multiple-Query Optimization. *ACM TODS* (13) 1 (1988).

[17] Sellis, T., Lin, C-C. and Raschid, L. Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms. Tech. Report CS-TR-1960, Dept. of Computer Science, University of Maryland, College Park (1987).

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

[19] Stonebraker, M., Sellis, T. and Hanson, E. Rule Indexing Implementations in Database Systems. *In [14].*

[20] Stonebraker, M. and Rowe, L. The Design of POSTGRES. *Proc. of the ACM-SIGMOD Conf.*, Washington, DC (1986).