An Algorithmic Basis for Integrating Production Systems and Large Databases

Daniel P. Miranker
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

David A. Brant
Applied Research Laboratories
The University of Texas at Austin
P.O. Box 8029
Austin, TX 78713-8029

ABSTRACT

Due to the similarities between AI production rules and relational database queries with updates, it appears straightforward to integrate the two systems to form an *active database system*. However, a large rule system represents on the order of hundreds or thousands of concurrent transactions, each repeated on every cycle. Executing such a large number of transactions on a large database within a short amount of time is computationally stressful. Main-memory resident production systems have been made computationally feasible by the development of incremental match algorithms that exploit the temporal redundancy of the database by saving results computed in prior cycles. Unfortunately, the worst-case space complexity of these match algorithms is exponential and space management becomes a dominant issue. In this paper we present a lazy incremental match algorithm with linear worst-case space complexity. Moreover, initial empirical results show that we prune 60% of the search for rule instantiations. We feel that these results provide the first reasonable algorithmic basis upon which one can develop an active database system.

1.0 Introduction

There is a large and growing body of research directed toward the integration of relational database and expert system technologies. Simple rules for enforcing database constraints or monitoring the database and automatically retrieving and updating data can be handled through alerters [1] and triggers [2]. Adaptations of view materialization [3,4] have been described for implementing restricted rule systems. More recently, Sellis, Lin, and Raschid have been developing a system which manages large rule bases using a relational database management system (DBMS) in a production system environment [5,6]. Perhaps the largest effort is the POSTGRES project [7]. Stonebraker and Rowe have reported extensively on their efforts to develop an INGRES successor having the ability to provide inferencing, with both forward and backward chaining. Work is also being done to address the architecture requirements of knowledge based systems [8-10]. These and other research projects have confirmed the extraordinary time and space demands one might expect from inferencing on large databases. While the average space requirements do not approach worst case, the variance in required space over the life of the system is often extremely large and unpredictable.

Our work focuses on the problem of using the production system paradigm as the deductive component of an expert database system. One of the fundamental issues is the exponential worst-case space requirement inherent in existing production system match algorithms [6,11,12]. Although worst case is rarely (if ever) achieved, it is entirely possible for such algorithms to unexpectedly exhaust all of the available storage in large virtual memory computer systems [9]. Therefore, before dealing directly with performance issues related to inferencing on large databases, our current work develops an algorithmic basis for matching that is better than current match algorithms in its space requirements. Toward that end, we have developed a new match algorithm which has a linear worst-case space complexity. This new algorithm serves as the basis for our investigations into an active database system. Before describing the algorithm, production systems and their relational interpretation are presented in section 2. Section 3 describes the new match algorithm and section 4 briefly describes an architecture to support the algorithm. Section 5 summarizes our current work.

2.0 Production Systems and Relational Databases

In general, a production system is defined by a set of rules, or productions, that form the production memory, together with a database of current assertions, called the working memory (WM). Each production has two parts, the left-hand side (LHS) and the right-hand side (RHS). The LHS contains a conjunction of pattern elements, or condition elements, that are matched against the working memory. The RHS contains directives that update the working memory by adding or deleting facts, and directives that carry out external side effects such as I/O. In operation, a production system interpreter repeats the following recognize-act cycle:

(1) Match. For each rule, compare the LHS against the current WM. Each subset of WM elements satisfying a rule's LHS is called an instantiation. All instantiations are enumerated to form the conflict set.

(2) Select. From the conflict set, chose a subset of instantiations according to some predefined criteria. In practice a single instantiation is selected from the conflict set on the basis of the recency, specificity, and/or rule priority of the matched data in the WM.

(3) Act. Execute the actions in the RHS of the rules indicated by the selected instantiations.

CH2840-7/0000/0353$01.00 © 1990 IEEE

Production systems can be viewed from a relational database (RDB) perspective, in much the same way that logic programming can be mapped to relational databases [13]. This observation allows a single framework within which both production systems and RDBs can be discussed.

2.1 Working Memory as a Relational Database

A working memory element (WME) forms the user's conceptual view of an object and consists of a class name followed by a list of attribute-value pairs. A class name identifies an object and the attribute-value pairs describe a particular instance of that object. Each WME has a unique identifier (ID) associated with it. IDs are often implemented as a strictly increasing sequence of integers assigned when the WME was created or last modified. They may be construed as timestamps or as logical pointers to individual WMEs. In most production systems, IDs are used in the conflict set resolution criteria. Consider the WME, shown below, used to describe a red cube named c_1, with a mass of 100, and having a length of 10 (attributes names are distinguished by a preceding ^ operator).

(cube ^name c_1 ^color red ^mass 100 ^len 10)

A simple mapping of WMEs to an RDB can be made by interpreting a class name as a relation name and the attribute names within a class as attribute names in the respective relation. The resulting relation is then augmented with an attribute corresponding to the ID. Thus, arbitrarily assuming an ID value of 506, the particular instance of the cube object above would be represented as the tuple <506,c_1,red,100,10> in the relation cube(ID,name,color,mass,len).

2.2 Productions as Relational Queries

A production's LHS consists of a conjunction of condition elements (CEs). It contains one or more non-negated CEs and zero or more negated CEs. Negated CEs are distinguished by a preceding negative sign. The LHS is said to be satisfied when:

(1) for each non-negated CE, there exists at least one matching WME, and,

(2) for all negated CEs, there do not exist any matching WMEs.

Each CE consists of a class name and one or more terms. Each term specifies an attribute within the class and a predicate to be evaluated against the values of that attribute. A CE need not reference all of the attributes contained in its corresponding class The class is projected onto the named attributes in the CE. Those not named do not affect the match criteria. Predicates consist of a comparison operator (<,>,=,≤,≥, or ≠) followed by a constant or variable. A predicate containing a constant is true with respect to a WME if the corresponding attribute value in the WME matches the predicate. For example, consider the CEs and corresponding WMEs shown in Fig. 1.

CE (a) matches WMEs (1) and (3), while CE (b) matches only WME (1). Constants within condition elements can be mapped to relational SELECT operations (σ).

CEs	"cube" WMEs			
	name	color	mass	len
a) (cube ^mass<10)	1) c_1	red	6	8
b) (cube ^mass<10 ^len>5)	2) c_2	blue	11	5
	3) c_3	red	1	3

Figure 1. Predicate Matching

The scope of a variable is the production in which it appears, and, therefore, all occurrences of a variable within a given LHS must be bound to the same value in working memory for the LHS to be satisfied. For condition elements containing variables, a mapping can be made to a relational JOIN operation (⊗). The JOIN operator will ensure that a given variable is consistently bound for all of its occurrences within a LHS.

To further describe the mapping of LHSs to relational algebra, the concept of CE dependence is introduced. A set of CEs within a given rule are said to be dependent if they share variable names. The instantiations for a production consist of the Cartesian product (×) of the results returned by the subqueries corresponding to the sets of dependent condition elements. For example, consider the mapping of the production containing two sets of dependent condition elements $\{C_0,C_1\}$ and $\{C_2, C_3\}$ shown in Fig. 2.

Production Rule:

(P Example
(C_0 ^A_0<12 ^A_1<x>)
(C_1 ^A_1<x>)
(C_2 ^A_2<y>)
(C_3 ^A_2<y> ^A_3>7)
---> *actions*)

Corresponding Query:

$$((\sigma_{A_0<12}C_0) \underset{(C_0.A_1=C_1.A_1)}{\otimes} C_1) \times (C_2 \underset{(C_2.A_2=C_3.A_2)}{\otimes} (\sigma_{A_3>7}C_3))$$

Figure 2. Mapping Dependent CEs

To simplify the mapping of negated CEs we introduce a new operator called NOT-JOIN (¬⊗). It is defined as $R \neg\otimes S = R - \pi_R(R \otimes S) = R - (R \text{ SEMIJOIN } S)$.

Production Rule:

(P Example
(C_0 ^A_0<x>)
(C_1^A_0<x> ^A_1<y>)
-(C_2 ^A_1<y>)
---> *actions*)

Corresponding Query:

$$(C_0 \underset{(C_0.A_0=C_1.A_0)}{\otimes} (C_1 \underset{(C_1.A_1=C_2.A_1)}{\neg\otimes} C_2))$$

Figure 3. Mapping Negated CEs

Using this operator we can map LHSs containing negated CEs (see Fig. 3).

2.3 Current Matching Algorithms

A naive algorithm for finding the instantiations of the rules (i.e., matching) would execute the query associated with each rule's LHS against the entire database on each cycle. That approach is combinatorially explosive and computationally intractable. However, the database in a production system is *temporally redundant*; i.e., on each cycle only a small subset of the working memory changes. Rather than reverify the satisfaction of each rule on every cycle, production system interpreters use incremental match algorithms. An incremental match algorithm maintains the results of the previous cycle's match phase (the joins and Cartesian products) and computes only the incremental change to the conflict set that results from the incremental change to the database. Consider the rule shown in Fig. 3. If a new tuple, T, is added to the relation C_0 then the following query can be used to find all new instantiations created by that tuple and update the conflict set.

$$\text{New\_Conflict\_Set} := \text{Conflict\_Set} \cup (T \underset{(T.A_0=C_1.A_0)}{\otimes} (C_1 \underset{(C_1.A_1=C_2.A_1)}{\neg\otimes} C_2)).$$

Updating the conflict set when deleting T from C_0 is accomplished by

$$\text{New\_Conflict\_Set} := \text{Conflict\_Set} - (T \underset{(T.A_0=C_1.A_0)}{\otimes} (C_1 \underset{(C_1.A_1=C_2.A_1)}{\neg\otimes} C_2)).$$

Several incremental match algorithms appear in the literature [6,11,12]. All of these algorithms trade space for time and have a worst-case space complexity of at least $O(n^c)$, where n is the size of the database and c is the maximum number of conditions in a rule. Although the average-case behavior is much better, it is still impractical if not impossible to store the internal state of these match algorithms for large databases.

The semantics of production systems, in conjunction with the operation of currently accepted production system match algorithms, demand that all instantiations of all rules be enumerated before one instantiation is selected for firing. The conflict set itself has a worst-case space of $O(n^c)$ and a highly volatile average space requirement. Tables 1, 2, and 3 show statistics for some OPS5 benchmarks presented in the literature [8,11]. From Tables 1 and 2 it can be seen that the size of the conflict set and its standard deviation are significant when compared to the working memory size [14]. Further, instantiations are often computed, entered into the conflict set, and subsequently removed, without ever firing. In a strong sense, the time and space required to compute and store these instantiations is wasted. An algorithm that avoids enumerating instantiations may avoid much of the wasted computation and space. Some initial results have been gathered on the effectiveness of lazy matching at pruning the search space. These are shown in Table 3.

Table 1. Working Memory Statistics

Program	Max. WM Size	Avg. WM Size	Std. Dev.
WALTZ	50	42.29	10.34
ROBOT	17	15.2	2.6
TOURNEY	279	123.1	44.66
JIG25	100	50.12	29.82
MESGEN	38	33.71	1.96

Table 2. Conflict Set Statistics

Program	Max. CS Size	Avg. CS Size	Std. Dev.
WALTZ	66	11.43	14.76
ROBOT	13	4.58	2.93
TOURNEY	881	138.30	203.47
JIG25	169	88.41	49.85
MESGEN	14	3.82	3.97

Table 3. Search Pruning

Program	Inst-antiations	Rule Firings	Unused Inst-antiations	% Unused
WALTZ	151	70	81	54
ROBOT	478	410	68	14
TOURNEY	2324	528	1796	77
JIG25	205	58	147	72
MESGEN	860	138	722	84

For the applications analyzed, an average of 60% of the instantiations computed by TREAT or RETE are never fired. The next section describes a "lazy" way to match rules in a production system. The idea is similar to the lazy evaluation used in functional programming languages; i.e., a function calculates its values only as they are needed. The Lazy Match has a worst-case space characteristic of $O(n*c)$.

3.0 A Lazy Matching Algorithm

The following describes a method for computing production instantiations in a lazy manner. This is accomplished by executing a best-first search for instantiations. Since the database may change from cycle to cycle, the best-first search must be capable of responding to a dynamic search space. The initial requirements for lazy matching are:

(1) maintain a total ordering in the generation of an instantiation, and

(2) ensure that a given instantiation is fired only once.

If the total ordering in the first requirement is by timestamp (i.e. ID), a search heuristic based upon firing the production with the most recent instantiation [15] can be employed. However, it is important to note that any total ordering of instantiations for a given rule will work. Adding additional criteria for instantiations of different rules, such as rule priority, can also be accommodated in a straightforward manner. The second requirement prevents simple cycles and creation of duplicate facts. In OPS5 these requirements are met by the conflict set resolution

strategy — LEX or MEA [16]. Since both are quite similar only LEX is described further.

3.1 LEX Conflict Set Resolution

The LEX strategy orders instantiations by successively comparing the recency of all data elements within them. Each instantiation is ordered by timestamp and pairs of elements from each are compared. This continues until it finds a data element in one that is more recent than the data element in the other. It then prefers the instantiation containing the more recent element. If one instantiation is exhausted before the other without finding a more recent element, then the one not exhausted is preferred. If both are exhausted at the same time, then the specificity of their corresponding LHSs is compared. The one containing more tests for constants and variables is preferred. If no single instantiation dominates at this point then an arbitrary selection is made from among the preferred instantiations.

3.2 Conflict Set Resolution in Lazy Matching

The challenge of the lazy matching algorithm is in controlling a best-first search for instantiations through a database that may change after each instantiation is found and fired. The criteria for "best" in this case is based upon the conflict set resolution strategies.

Lazy matching uses the selection strategy as an evaluating function to direct the search for a firable instance. This is done by using that criteria to direct the search for matching tuples from the *alpha-relations* of the database which correspond to the *alpha-memories* in RETE and TREAT. On any given cycle, the search for an instantiation will stop after the first one is found, with the search being conducted so as to preserve recency. Of course, additions to, and deletions from, the database will affect the search, and we must ensure that a given instantiation is fired at most once. To do so, state information of some form must be saved from cycle to cycle in order to continue the correct computation of instantiations.

3.3 Computing Instantiations Using Lazy Matching

The concept of a dominant tuple (DT) is introduced to control the lazy computation of instantiations. Each tuple will be selected from the database as a DT exactly once. To enforce this requirement, the relational mapping described earlier is augmented with a boolean attribute (S) to indicate whether or not a tuple has been selected as a DT. All tuples are initially set with S=0. The DT is that tuple not already marked as "selected" (i.e., S=0) and containing the most recent timestamp (i.e., the maximum ts). Figure 4(a) shows the initial database for the following query. Note that the ID in this example is a timestamp denoted ts.

(P Example
(R_0 ^A<x>)
(R_1 ^A<x> ^B<y>)
(R_2 ^B<y>)
---> *actions...*)

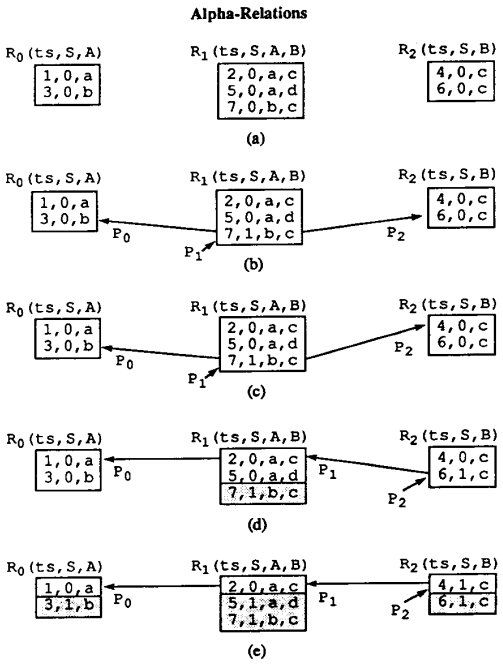

Figure 4. Computing Instantiations

The computation of an instantiation begins with selecting the DT, and marking it as selected; i.e., S:=1. This is followed by a best-first search for an instantiation containing DT. To ensure that instantiations are produced only once, relations have a fixed ordering (by timestamp in this example), and the best-first search computation restricts the tuples joining with DT to those having timestamps less than that of the dominant tuple. As soon as a matching set of tuples (i.e., an instantiation) for DT is found, the computation pauses and the result is fired. If an instantiation containing DT cannot be found, then a new DT is chosen, marked as selected, and a new best-first search is begun. If all DTs are exhausted without finding an instantiation, then the system halts. Figure 4(b) shows the initial state of the best-first search pointers (P_i). In this example, the best-first search is rooted at ts=7 in relation R_1 and proceeds outward in join order, most recent to least recent tuple in each relation. The timestamps are used as logical pointers for the search. The state of any search is represented as a tuple of timestamps — one from each relation. Thus, for Fig. 4(b) the search state is «3,7,6». These tuples satisfy the query and thus become the first instantiation. Next, the rule is fired, and, assuming for now that no tuples are added to or removed from the database by firing the rule, the search resumes to find the next instantiation. Figure 4(c) shows the state of the search after finding the next instantiation — «3,7,4». Before finding «3,7,4» the search would have tried «1,7,6», failed, backtracked, advanced the P_2 pointer, and succeeded (we arbitrarily chose to search the left relation first). The next time the search is performed «1,7,4» will be

tried and will fail. That will exhaust the search rooted at the DT with ts=7. At that time a new DT must be chosen. In this case it is the tuple with ts=6. The shaded area in Fig. 4(d) contains tuples that have timestamps greater than that of the DT and therefore are not considered in the search. The next instantiation to be found is «1,2,6», after unsuccessfully trying «3,5,6», «1,5,6», and «3,2,6». After that, the tuple with ts=4 is chosen as DT and «1,2,4» is found (Fig. 4(e)) after trying «3,5,4», «1,5,4», and «3,2,4». «1,2,4» is the final instantiation that can be produced. After it is fired all DT searches will have been exhausted and no new instantiations can be found. Thus, the system halts.

We now consider the effects of adding and deleting tuples after each rule firing. To do so we introduce a stack. Each element on the stack represents the state of a suspended best-first search. When a new tuple is added to the database the current search is suspended and its state pushed onto the stack. That search will be resumed at a later time when its DT is again the most recent. Since deletions may affect the state of a suspended search by removing tuples that have pointers to them on the stack, each time a search state is popped from the stack, its pointers must be verified by the best-first search. If the DT has been deleted then its search is exhausted and a new DT chosen. If any other tuple has been deleted, the search backtracks to find the next instantiation. If none is found then again the search is exhausted.

Figure 5(a) is the same as Fig. 4(b) except that a stack has been added. Assume that the instantiation referenced by «3,7,6» fires and adds the tuple <8,0,d> to R_2. This causes

(1) the search state «3,7,6» to be pushed to the stack,

(2) <8,0,d> to be chosen as the DT (thus changed to <8,1,d>), and

(3) the next instantiation to be found, i.e., «1,5,8» (Fig. 5 (b)).

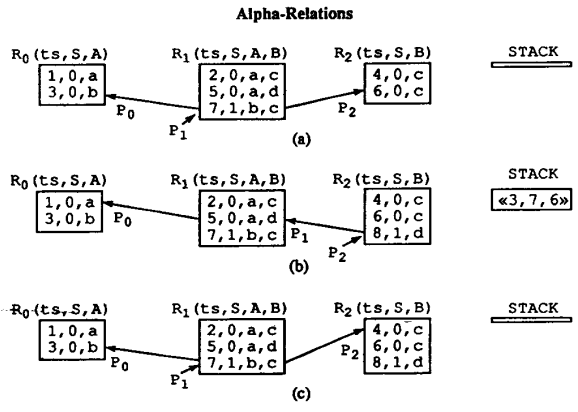

Figure 5. Dynamic Search Space

Assume that firing «1,5,8» does not change the database. On the next cycle the search rooted at ts=8 will be exhausted and the top of stack DT, 7, will be compared to the most recent database tuple that has S=0; 6 in this case. Since the stack tuple is more recent, its search will be resumed. The next instantiation found will be «3,7,4». The pseudocode for Lazy Match in Fig. 6 should help elucidate the algorithm.

There are pathological cases where the best-first search strategy will not produce the identical sequence of instantiations as OPS5; nevertheless the criteria used in lazy matching is in keeping with the general concept of recency as presented by McDermott and Forgy [15], and has not posed a problem in two large systems.

This discussion has considered the generation of an instantiation by the best-first search as a computation involving a single rule. The algorithm can be extended to multiple rule systems in several ways. The most interesting approach is to execute the algorithm independently for each rule and form a limited conflict set with at most one instantiation from each rule. Then use any combination of rule priority, recency, and/or specificity to select the instantiation to be fired. This also introduces rule-level parallelism that can be exploited on parallel processors.

3.4 Handling Negated Condition Elements

We have discovered three different methods of lazily handling negated condition elements (NCEs). Only one will be described here. The methods for dealing with NCEs are closely related to the method developed for the TREAT match algorithm [11]. If a search for an instantiation consistently binds with a tuple that matches an NCE, then the search fails at that point and must backtrack. We say that tuple blocked the search. When a blocking tuple is removed from the system, some instantiations may become unblocked and allowed to compete for firing. Those instantiations that become unblocked are those that would have been computed had the condition element been positive instead of negative, and had the tuple been added to the system instead of removed.

To handle NCEs, for each negated condition, C_i, add a second alpha-relation which will *shadow* the first. Rename the original alpha-relation from R_i to R_i^-. R_i, as before, contains the tuples that match the constants in C_i. Call the shadow alpha-relation R_i^S. When a tuple that has blocked a search is removed from an R_i^- alpha-relation it is inserted into R_i^S and is given the next available timestamp. The timestamps of the tuples in R_i^S participate in the DB_DT.ts function. The newly added tuples to R_i^S can then be allowed to root a best-first search for those instantiations that they had blocked.

A problem arises when a search leads to an instantiation that has already been derived from a tuple in R_i^S. This is solved by requiring best_first_search to examine R_i^S. A search that starts with a DT timestamp ts_1 and binds consistently with a tuple in R_i^S with timestamp $ts_2 > ts_1$ fails. The idea is that once a tuple enters R_i^S, only it may generate instantiations with older tuples. Such a tuple will be able to root the

```
program LAZY MATCH;
R0,...,Rn-1: relation; {Alpha-relations corresponding
    to the working memory.}
P0,...,Pn-1: timestamp; {Pointers to tuples in the Ri —
    they are manipulated by the best-first search rou-
    tine and constitute a reference to an instantiation
    if one is found.}
stack: stack of n-tuples of pointers; {Stack elements
    are «P0,...,Pn-1», from suspended (superseded)
    best-first search computations.}
DT.ts: timestamp; {The timestamp of the dominant
    tuple that roots a best-first search.}
DT.S: boolean; {The selected bit for the dominant tu-
    ple.}
function DB_DT.ts return timestamp; {Function
    that returns the timestamp of the most recent tu-
    ple in the database that has not already been
    marked as selected, i.e., S=0. If no such times-
    tamp can be found then the result is 0.}
function stack_DT.ts return timestamp; {Function
    that returns the maximum timestamp of the
    pointers, P0,...,Pn-1, on the top of stack. If the
    stack is empty result is 0.}
procedure push(P0,...,Pn-1: in timestamp);
    {Procedure that pushes the best-first search point-
    ers, P0,...,Pn-1, onto the stack.}
procedure pop(P0,...,Pn-1: out timestamp);
    {Procedure that pops pointers, P0,...,Pn-1, off of
    the stack.}
procedure best_first_search(P0,...,Pn-1: in out
    timestamp; found: out boolean); {Procedure that
    performs a best-first search for an instantiation
    beginning at P0,...,Pn-1, and returns a new
    P0,...,Pn-1. Before starting the search, the point-
    ers are checked to ensure that the tuples they
    point to have note been deleted. If no instantiation
    is found using the DT.ts in the input P0,...,Pn-1,
    then found=false. The search does not consider
    any tuples with a ts>DT.ts. The search is rooted
    at the tuple referenced by DT.ts. It branches out-
    ward to the other relations as specified by the join
    order for the rule. Each relation is processed in
    timestamp order, with most recent first. If a rela-
    tion is searched and no matching tuple is found,
    the search backtracks to the previous relation. If
    it backtracks to the root, i.e., the relation contain-
    ing DT.ts, then the search is said to be exhausted
    for that DT.ts and the procedure returns with
    found=false.}
procedure fire(P0,...,Pn-1: in timestamp);
    {Procedure that fires the instantiation referenced
    by the pointers, P0,...,Pn-1.}
begin
    initialize Ri; {ordered by timestamp and S=false}
    initialize Pi; {∀ i, Pi:=0}
    initialize stack; {to empty}
    DT.ts:= 0;
    loop
        {Find out if the database contains the domi-
        nant tuple. This will be true if there were any
        additions on the previous cycle, or if the cur-
        rent DT search was exhausted and the next
        most recent tuple is in the database.}
    if (DB_DT.ts>DT.ts ) AND
        (DB_DT.ts>stack_DT.ts) then
        if DT.ts≠0 then push(P0,...,Pn-1); end if;
            DT.ts:= DB_DT.ts;
            DT.S:= true;
            ∀ i, Pi:= DT.ts; {this sets the initial starting
                point for any new best-first search. Best-
                first search will return the first set of
                valid pointers reached from this point by
                backtracking.}
        elsif stack_DT.ts>DT.ts then
            {If DT does not come from the database,
            find out if the stack has the most recent
            DT.ts. This will only be true if the current
            DT.ts=0 and the stack is not empty.}
            DT.ts:=stack_DT.ts;
            pop(P0,...,Pn-1);
        end if;
        {we now have the timestamp of the dominant
        tuple and can proceed to find an instantiation,
        or stop if DT.ts=0}
        exit when DT.ts=0;
        best_first_search(P0,...,Pn-1,found);
        if found then
            fire(P0,...,Pn-1);
        else
            DT.ts:=0;
        end if;
    end loop;
end.
```

Figure 6. Lazy Match Pseudocode

search for all instantiations older than itself, whether they were blocked or not.

3.5 Algorithmic Complexity

A tuple can not exist in both an R_i relation and it's shadow, R_i^S, at the same time. Shadow relations may be implemented by augmenting the R_i relations with an additional boolean attribute representing if a tuple has been deleted. So implemented, the worst-case time complexity of Lazy Match is $O(n^c)$, the same as for RETE and TREAT.

To determine the space complexity of Lazy Match for p rules, consider that each tuple can appear as a dominant tuple at most once. Thus, the size of the stack is bounded by the size of the database, as is the size of each alpha-relation. For each rule there is a maximum of c condition elements. The worst-case space complexity of Lazy Match is $O(p*n*c)$.

We anticipate that overall speed of the algorithm will exceed TREAT, even in main-memory systems. For

a given program the searches executed by Lazy Match are a subset of the searches executed by TREAT, and, except for some overhead, those searches are performed by executing precisely the same sequence of instructions. In comparison to the total execution time of the system the additional overhead will not be significant.

Since Lazy Match must maintain a stack of suspended computations it is possible that it will require more space than the TREAT algorithm. This will more likely be true for programs with small maximum conflict set size. However, the amount of space required for Lazy Match will be consistent cycle to cycle.

4.0 An Architecture to Support Lazy Matching

The Lazy Match algorithm provides a stable basis to build on, but does not fully address performance issues. Parallelism appears to be one worthwhile approach to dealing with the computational requirements of these systems [6,11,12]. A parallel architecture to support I/O intensive applications (a.k.a., the KYKLOS Database Engine) is being investigated as to its suitability for both relational databases and the system described in this paper [17,18]. The architecture lends itself to parallel access of databases, and to parallel operations on data objects. It is being emulated on the Symult 2010 parallel processor computer. The current configuration contains 24 processors and eight disk drives which can be accessed in parallel. Database algorithms tailored to this environment are being developed. The key is to distribute the database over the eight I/O nodes such that the majority of disk accesses can be made concurrently.

Our approach is to not only process the production rules concurrently (rule-level parallelism), but to take advantage of the high degree of data-level parallelism available in a database environment. The copy-and-constrain technique [19] is an example of an attempt to exploit data-level parallelism. Our approach is similar except that we segment the database instead of adding patterns. Each I/O node contains only a portion of the database. A copy of all rules resides at each I/O node and is allowed to match on data available at that node. This work is immature, but the prospects are encouraging.

5.0 Current Research

Current work is focusing on the definition of an appropriate language and compiler for database oriented production systems, an analysis of the trade-offs involved in using rule-level parallelism, and an implementation of the system on the Symult 2010 parallel computer. The compiler may include an analysis stage that determines whether a rule should be evaluated lazily or eagerly [20]. The effect of the eager evaluation will be to produce set oriented operations which can be efficiently computed on the Symult machine. The trade-off analysis in specifying the details of the Lazy Match algorithm will be done through the analysis of several applications. To help understand the effect of WM size on performance, the test applications are being scaled up in the number of WMEs.

Also under development is a disk-based implementation of an OPS5 compiler. The basis for this work is the Jupiter file management system being developed at The University of Texas at Austin. Jupiter and its higher level companion, Genesis, allow one to build reconfigurable DBMSs using a "building block" approach [21]. By exploiting this technology, we have the capability to rapidly perform many experiments on the construction of active database systems.

6.0 Conclusion

Before the effective integration of production systems and large databases can be achieved, the exponential worst-case space complexity of the match algorithms currently in use must be dealt with. The space usage of these algorithms is directly attributable to their "eager" nature; i.e., they require that all instantiations be enumerated before one is chosen to fire. We have described a "lazy" match algorithm which has a linear worst-case space complexity. We believe this result to be a necessary foundation upon which to build viable database oriented production systems. Moreover, by only producing instantiations as they are needed, the Lazy Match has been shown to prune approximately 60% of the search space on sample applications. We are working toward a parallel implementation of an active database system based on the KYKLOS Database Engine architecture. The goal is to take maximum advantage of both rule-level and data-level parallelism in a stable environment.

REFERENCES

[1] Bunemann, P. and E. Clemons, "Efficiently Monitoring Relational Data Bases," ACM-TODS, Sept. 1979.

[2] Astrhan, M., et. al., "System R: A Relational Approach to Data," ACM-TODS, June 1976.

[3] Ullman, J., "Implementation of Logical Query Languages for Data Bases," Proceedings of the 1985 ACM-SIGMOD International Conference on Management of Data, Austin, TX, May 1985.

[4] Blakeley, J. A., et. al., "Efficiently Updating Materialized Views," Proceedings of the 1986 ACM-SIGMOD International Conference on Management of Data, Washington, DC, June 1986.

[5] Raschid, L., and S. Su, "A Transaction Oriented Mechanism to Control Processing in a Knowledge Base Management System," Proceedings of the Second International Conference on Expert Database Systems, 1988.

[6] Raschid, L., T. Sellis, and C-C Lin, "Exploiting Concurrency in a DBMS Implementation for Production Systems," Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, 1988.

[7] Stonebraker, M., "The Design of the POSTGRES Storage System," Proceedings of the 13th VLDB Conference, Brighton, 1987.

[8] Gupta, A., C. Forgy, and A. Newell, "High-Speed Implementations of Rule-Based Systems," ACM TOCS, June, 1989.

[9] Bein, J., R. King, and N. Kamel, "MOBY: An Architecture for Distributed Expert Database Systems," Proceedings of the 13th VLDB Conference, Brighton, 1987.

[10]Stolfo, S., and D. Miranker, "DADO: A Parallel Processor for Expert Systems," Proceedings of the International Conference on Parallel Processing, pp. 74-82, 1984.

[11]Miranker, D., "TREAT: A Better Match Algorithm for AI Production Systems," Proceedings of the 1987 National Conference on Artificial Intelligence , Seattle, 1987.

[12]Forgy, C., "RETE: A Fast Match Algorithm for the Many Pattern/Many Object Pattern Match Problem," Artificial Intelligence, no. 19, pp. 17-37, 1982.

[13]Nicolas, J., "Logic and Databases," in Logic and Databases, H. Gallaire and J. Minker (eds.), Plenum Press, NewYork, 1978.

[14]Lofaso, B. J., "Join Optimization in a Compiled OPS5 Environment," Tech. Report No. ARL-TR-89-19, Applied Research Laboratories, The University of Texas at Austin, April, 1989.

[15]McDermott J., and C. Forgy, "Production System Conflict Resolution Strategies," In Pattern-directed Inference Systems, D. Waterman and F. Hayes-Roth (eds.), Academic Press, 1978.

[16]Forgy, C., "OPS5 User's Manual", Tech Report CMU-CS-81-135, Carnegie-Mellon University, 1981.

[17]Dale, A.G., F. Haddix, R. Jenevein, and C.B. Walton, "Scalability of Parallel Joins on High Performance Multicomputers," Tech. Report No. TR-89-17, Dept. of Computer Sciences, University of Texas at Austin, June, 1989.

[18]Brant, D., B. Menezes, D. Loewi, A. Dale, R. Jenevein, "An Interconnection Network to Support Relational Database Join Operations," Proceedings of the Seventh International Conference on Distributed Computer Systems, 1987.

[19]Stolfo, S., D. P. Miranker, and R. Mills, "A Simple Processing Scheme to Extract and Balance Implicit Parallelism in the Concurrent Match of Production Rules," IFIP Conference on Fifth Generation Computing, 1985.

[20]Kuo, C. M., D. P. Miranker, and J. C. Browne, "A Concurrent Rule Execution Language, CREL, and CREL Parallelizing Transforms," Tech. Report, Dept. of Computer Sciences, University of Texas at Austin, October, 1989.

[21]Batory, D., et. al., "GENESIS: An Extensible Database Management System," IEEE Transactions on Software Engineering, Nov., 1988.