TOWARD AN INTEGRATED
EXPERT DATABASE SYSTEM*

DANIEL P. MIRANKER
DAVID A. BRANT

MARCH 1989 AI89-100

* This research was supported in part by the Office of Naval Research under contract N00014-86-K-0763 and by a grant from Texas Instruments.
Toward an Integrated Expert Database System

by

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

Incremental match algorithms such as RETE and TREAT have an exponential worst-case space complexity. Thus, it is impractical, using currently known methods, for an expert-system written in production system form to directly absorb the quantity of data present in even small databases. Therefore, expert-system applications with a database component internally maintain a distinct copy of a small subset of the database and poll the database when new information is needed. We describe a "lazy" match algorithm that has linear worst case space complexity in the size of the database and will prune much of the search for instantiated rules. We consider the lazy match to be the prerequisite result for developing integrated rule-based expert database systems.
1. Introduction

Many expert system applications require access to very large existing databases. Examples of such applications include telephone cable maintenance [Vesonder83], credit-card fraud detection, embezzlement investigation, and derivation of the fare structure in the airline industry.

Production rule systems are a dominant method for codifying expertise. 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 affect 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 of the matched data in the WM.

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

A naïve algorithm for checking the satisfaction of the rules in a system would check each clause, or condition, of each rule against the entire working memory on each cycle. The naïve approach is combinatorially explosive and computationally intractable. However, the working memory in a production system is temporally redundant. That is, 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 state across production system cycles and computes an incremental change to the conflict set as the result of an incremental change to the working memory.

Two incremental match algorithms, TREAT [Miranker87] and RETE [Forgy82] appear in the literature. Both algorithms trade space for time and have worst-case space complexity of $O(n^c)$, where $n$ is the size of the working memory and $c$ is the number of
conditions in a rule. Although the average-case behavior is much better, it is still
impracticle if not impossible to store the internal state of these match algorithms.

A common solution to this problem is to study these two systems in isolation and
develop methods for interfacing them [vanBuer85, Moto-oka84, Kellogg81]. In this
approach the database becomes a passive element of the system. Changes to the database
that may need the immediate attention of the expert system will not be noticed until the
expert system queries the database (i.e., the behavior is polled instead of interrupt driven.)
Even when polling the database the amount of data that is returned to the expert system is
often unpredictable. If the amount of data is unexpectedly large, the response time of the
system may become intolerably slow, and the maximum working store of the expert system
may be exceeded, resulting in the complete failure of the system.

This paper describes a preliminary work toward an integrated expert database system.
The goals include:

- Alleviating the communications bottleneck by having the expert system and database
  system coexist on the same machine and having both operate on the same data
  structures.

- Making the database, and transactions on it, active components of the expert system.

An integral part of the system is the development of a "lazy" match algorithm. 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). 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.
Section 3 describes a "lazy" way to match the 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 by the calling program. The lazy match has a
worst-case space characteristic of O(n^c).

An effective implementation of the the lazy match will not be completely lazy.
Production system programs display a high degree of locality [Gupta86, Miranker87].
Locality is already exploited unecessary computation in both TREAT and RETE
[McDermott77, Miranker87]. In lazy matching locality may be used to determine whether
any search is called for at all. Thus, part of the implementation of the lazy match is will
be quite eager. We expect the eager part of the match to be easily computable and the lazy match to be useful for ordinary production systems on sequential machines. For large databases the expert system will need rapid access to the database and the computational resources to support the eager phase. Thus, for the integrated expert database system we presume the existence of parallel I/O Engine. Section 4 describes the configuration of the Symult 2010, a parallel computer with multiple I/O sites that we have at The University of Texas at Austin. This machine in conjunction with work on the KYKLOS-II I/O Engine [Jenevein86] will serve as underlying architectures for the investigation of our system.

Before examining the lazy match algorithm it is helpful to have a common basis for describing both databases and production systems. Toward that end, the next section presents a simple mapping of production systems to relational databases.

2. Mapping Production Systems to Relational Databases

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 [Nicolas78]. This observation allows a single framework within which both production systems and RDBs can be discussed.

2.1 Working Memory as a Relational Database

Working memory elements (WMEs) consist of class names followed by a list of attribute-value pairs. A simple mapping of WMEs to an RDB can be made by interpreting class names as relation names and the attribute names within a class as attribute names in the respective relation. To illustrate this mapping, consider the OPS5 production in Example 1 and the corresponding relations to its right. The notation used expresses a relation with n attributes as [Ullman82]:

Relation_Name(Attribute_Name0,Attribute_Name1,...,Attribute_Name_n)

Notice the inclusion of the attribute named TID (Tuple IDentifier) in each relation. A TID contains a system assigned unique timestamp (TID.TS) and a tag bit (TID.S_bit) used by the lazy matching algorithm. When a tuple is added to the database, TID.TS := f(I+) and TID.S_bit := 0. f(I+) is a monotonically increasing function that returns positive integers, e.g., an integer counter.

(P Example)
(Class_1 ^attr_1 Const_1 ^attr_2 <x>) => Class_1(TID,attr_1,attr_2)
(Class_2 ^attr_2 <x> ^attr_3 Const_2) => Class_2(TID,attr_2,attr_3)
-(Class_3 ^attr_3 <x>) => Class_3(TID,attr_2)
--->...

Example 1.
2.2 Productions as Relational Queries

Constants within condition elements map to relational select operations. For condition elements containing variables, a mapping can be made to a relational join operation. A set of condition elements are said to be dependent if they share variable names. The join operator will ensure that those variables are bound consistently. The instantiations for a production consist of the Cartesian product of the results returned by the queries corresponding to the sets of dependent condition elements. For example, consider the mapping of the following production containing two sets of dependent condition elements – \{C_0, C_1\} and \{C_2, C_3, C_4\}. The corresponding Cartesian product is shown on the right.

(P Example_2
\(C_0 \wedge\text{attr}_0 \wedge\text{const}_0 \wedge\text{attr}_1 <x>\)
\(C_1 \wedge\text{attr}_1 <x>\)
\(C_2 \wedge\text{attr}_2 <y>\)
\(C_3 \wedge\text{attr}_2 <y> \wedge\text{attr}_3 <z>\)
\(C_4 \wedge\text{attr}_3 <z> \wedge\text{attr}_4 \wedge\text{const}_1\)
---\> ...

((SELECT \(C_0\)) JOIN \(C_1\))
\(X\)
(C_2 JOIN C_3 JOIN (SELECT \(C_4\)))

Example 2.

3. A Lazy Matching Algorithm

It is assumed that all relations discussed from this point on will contain the tuples that have been filtered by the select operations and thus correspond to the alpha-memories of RETE and TREAT. These are called alpha-relations.

Existing production systems require that all possible instantiations be enumerated before applying some criteria to select one instantiation to fire. When the working memory is large, as is the case with expert databases, the amount of computation time and storage space required by this approach is combinatorially explosive. Given a production system with \(c\) condition elements and a database having \(n\) tuples, the worst case time and space complexity per cycle is \(O(n^c)\) for RETE and TREAT based algorithms. However, only a small fraction of that time and space goes into computing the instantiation that is fired. If it were possible to compute that single instantiation directly, without enumerating the conflict set, then such a production system might significantly prune the search space and reduce the data access requirements. Thus, an effective expert database system might possible. The following sections describe a method for computing production instantiations in a lazy manner, i.e., at most one instantiation is computed on any one cycle. 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 some concept of recency in the generation of an instantiation, and
2. ensure that a given instantiation is fired only once.

The first requirement recognizes the effectiveness of a search heuristic based upon firing
the production with the most recent instantiation [McDermott78b]. The second prevents
simple cycles. In OPS5 these requirements are met by the conflict set resolution strategy –
LEX or MEA [Forgy81]. Since both are quite similar only LEX is described further.

3.1. LEX Conflict Set Resolution

The LEX strategy orders instantiations by successively pair-wise comparing the
recency of all data elements within them. 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 data space that may change after each instantiation is found. The
criteria for "best" in this case is based upon the conflict set resolution strategies. The
criteria used in LEX and MEA is a combination of recency and specificity, with recency
being dominant. The basic concept being explored in lazy matching is using the select
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. 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. A tuple can be selected from the database as a DT only once. The DT is that
tuple not already marked as "selected" (i.e., its TID.S_bit=0), and containing the most
recent timestamp (i.e., the maximum TID.TS).
The computation of an instantiation begins with the selection of DT, followed by a best-first search for an instantiation containing DT. The best-first search computation restricts the tuples joining with DT to have TID.TSs less than the dominant tuple TID.TS (called the DTID.TS). 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 1 illustrates these concepts for an equijoin over the common attribute names of the alpha-relations R0, R1, R2, and R3. The pointers associated with the best-first search are shown as Pi. A0 is the TID, with its two components, TID.TS and TID.S_bit, separated by a period( ). In this example, the best-first search is rooted at TID.TS=16 and ordered left to right, most recent to least recent. The pointers for the search are the TID.TSs of the tuples. Thus, for Fig. 1 the pointers are (16, 8, 12, 14). The shaded areas contain TID.TSs greater than DTID.TS=16 and therefore are not considered in the search rooted by DTID.TS=16. Note that all TID.S_bits=1 for TID.TSs in the shaded area. They have previously been chosen as the DT and their best-first search is completed or suspended. Also notice that some of the TID.TS that are in the valid search space (e.g., 10 and 12) have their TID.S_bit=1. This indicates that they have at one time been selected as the DT, and that following that, new tuples where added.

![Figure 1](image)

If firing adds no new tuples then the best-first search for the next instantiation is resumed. In Fig. 1, the next instantiation would be represented by (16, 8, 12, 10). If the RHS action of the fired rule adds new tuples to the alpha-relations, their TID.TSs will be greater than the current DTID.TS, and a new best-first search must begin using the most current tuple's TID.TS as the root. However, the existing best-first search computation may need to be resumed at some later point, should its DTID.TS become the most recent,
due to firings and/or deletions. To support this requirement, a stack of best-first search
pointers is maintained. If the search associated with the current DT is superceded then its
pointers are pushed onto the stack. The top-of-stack DTID.TS is allowed to compete
against the current DTID.TS for execution. When a search is resumed, some or all of the
tuples referenced by the pointers may have been deleted. To handle this condition, the
best-first search is given the last known set of pointers and backtracks if necessary to begin a
valid search. The following pseudocode should help elucidate the algorithm:
$R_0...R_{n-1}$: alpha-relations corresponding to the working memory, specifically the alpha memories in RETE and TREAT.

$P_0...P_{n-1}$: pointers (TID.TSs) to tuples in the $R_i$ - they are manipulated by the best-first search routine and constitute a reference to an instantiation if one is found.

$S$: stack of pointers, $P_0...P_{n-1}$, from suspended (superseded) best-first search computations.

DTID.TS: the TID.TS of the dominant tuple, DT, that roots a best-first search.

database_DTID.TS($R_i$): returns the TID.TS of the most recent tuple in the $R_i$ that has not already been marked as selected, i.e., TID.S_bit=0. If no such TID.TS can be found then the result is 0.

stack_DTID.TS($S$): returns the maximum $P_i$ of the pointers, $P_0...P_{n-1}$, on the top of stack. If the stack is empty result is 0.

push($P_0...P_{n-1}$): pushes the best-first search pointers, $P_0...P_{n-1}$, onto the stack.

pop($P_0...P_{n-1}$): pops pointers, $P_0...P_{n-1}$, off of the stack.

best_first_search($P_0...P_{n-1}$): function that performs a best-first search for an instantiation beginning at $P_0...P_{n-1}$, and returns a new $P_0...P_{n-1}$. If no instance is found using the DTID.TS in the input $P_0...P_{n-1}$, then $\forall i$ in the result, $P_i:=0$. The search does not consider any tuples with a TID.TS>DTID.TS.

Initialize $R_i$;

$\forall i, P_i:=0$;

initialize $S$;

DTID.TS:=0;

done:=false;

while NOT done do

{first check if there is has been an addition to the alpha-relations}

if (database_DTID.TS($R_i$) > DTID.TS) AND

(database_DTID.TS($R_i$) > stack_DTID.TS($S$)) then begin

if DTID.TS=0 then push($P_0...P_{n-1}$);

DTID.TS:=database_DTID.TS($R_i$);

$\forall i, P_i:=DTID.TS$; {this will cause best-first search to find the first set of valid pointers by backtracking}

end if;

{if no additions then check if the stack has the most recent DTID.TS -- this will only be true if the current DTID.TS=0 and the stack is not empty}

else if stack_DTID.TS($S$) > DTID.TS then begin

DTID.TS:=stack_DTID.TS($S$);

pop($P_0...P_{n-1}$)

end else;

{we now have the TID.TS of the most recent tuple available and can proceed to find an instantiation, or stop if that TID.TS=0}

if DTID.TS=0 then begin

best_first_search($P_0...P_{n-1}$);

if an instantiation is found then fire the new $P_0...P_{n-1}$;

else DTID.TS:=0;

end if;

else done:=true;

end while;

There are pathological cases where the best-first search strategy will not produce the identical sequence of instantiations as OPSS, nevertheless the criteria used in lazy matching
is in keeping with the general concept of recency as presented by McDermott and Forgy [McDermott78b], and it is guaranteed that the most recent tuple in the database is allowed to match and fire if possible.

This discussion has considered the generation of an instantiation by the best-first search as a global computation across all rules. Also under study is the possibility of executing the algorithm independently for each rule and forming a conflict set with at most one instantiation from each rule. This introduces rule-level parallelism and may be beneficial in some circumstances.

3.4. Algorithmic Complexity and Handling Negated Condition Elements

All of the algorithms mentioned in this paper make use of condition membership in the form of alpha-relations equivalent to the alpha-memories in RETE and TREAT. Assume the database (i.e., working memory) contains $n$ tuples and there are $c$ condition elements in the rule containing the maximum number of CEs. In this discussion of algorithmic space complexity we distinguish between the alpha-relations which have a worst case space of $O(n^c)$ and the remainder of the state space used to minimize the work. The RETE match saves both the conflict set and the partial join results (beta-memories). Each has an $O(n^c)$ worst case space complexity. TREAT saves considerable space by eliminating the beta-memories but still saves the conflict set. Though the conflict set is typically very small [Gupta86], TREAT still has an $O(n^c)$ worst case space complexity.

By producing instantiations best-first, the lazy match eliminates both the beta-memories and the conflict set. Thus the space requirement is proportional to the growth of the stack. The worst case space requirement occurs when every DT computation is suspended and its pointers are pushed onto the stack. The number of DTs is bounded by $n$, the number of tuples in the database. Thus, without considering negated condition elements, the worst case space is $O(n^c)$.

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. 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 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 partially match $C_i$. Call the shadow alpha-relation $R_i^+$. When a tuple that has blocked a search is removed from an $R_i^-$ alpha-relation it is inserted into $R_i^+$ and is given the next available TID.TS. The TID.TSs of the tuples in $R_i^+$ participate in the database_DTID.TS($R_i$) function. The newly added tuples to $R_i^+$ can then be allowed to root a best-first search for those instantiations that they blocked.

A problem arises when a search leads to an instantiation that has already been derived from a tuple in $R_i^+$. This is solved by requiring best_first_search($P_0...P_{n-1}$) to examine $R_i^+$. Searches that start with a DT, DTID.TS$_1$, and bind consistently with a tuple in $R_i^+$, TID.TS$_2$, such that DTID.TS$_1 <$ TID.TS$_2$ fail. The idea is that once a tuple enters $R_i^+$, only it may generate instantiations with older TID.TSs. Such a tuple will be able to root the search for all instantiations older than itself, whether they were blocked or not.$^1$

This method adds to the time of the algorithm, but is very simple to implement and its impact may be mitigated in both time and space by other simple filtering operations. Since the space required by the $R_i^+$ is bounded by the size of the database, the space complexity is not changed.

The other two methods for handling NCEs put the processing burden on the search rooted by tuples from the $R_i^+$. One is very simple to compute but requires $O(n^2)$ worst case space. The other has an $O(n^2)$ worst case space but is more complicated than the $O(n^c)$ solution. The actual performance and implementation of each approach is one current area of research.

4. An Architecture to Support Lazy Matching

A parallel architecture to support I/O intensive applications (a.k.a., the I/O Engine) is being investigated as to its suitability for both relational databases and the system described in this paper [Browne85, Jenevein86, Menezes85, Menezes86]. 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. Figure 2 shows the current configuration, in which eight disk drives 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.

$^1$ Note in this circumstance we agree with OPS's select definition which allows an instantiation to fire twice (or more) provided a negatively matching element has entered and left the system.
Even though the lazy matching algorithm can significantly decrease the time and space needed to process a large database, there may still remain a sizable amount of computational work to be done and data to be examined. To further improve the performance of such a system, parallelism can be introduced. Not only may the affect set of production rules be processed concurrently (rule-level parallelism), but the processing of each rule should take
advantage of the high degree of data-level parallelism available in a database environment. The copy-and-constrain technique [Stolfo85] is an example of an attempt to exploit data-level parallelism. Their approach was to make constrained copies of rules. Each copy referenced a subset of the data needed by the original rule. Our approach is similar but the constrained partitioning is done by segmenting the database instead of adding patterns. Each I/O node contains only a portion of the database. A copy all rule resides at each I/O node and is allowed to match on data available at that node.

Figure 3 shows the example of the data storage model that is used to support the lazy matching. Two separate schemas are stored – a conventional tuple based schema (TBS) and an attribute based schema (ABS). This not only provides fault tolerance, but also provides an economical means for precomputing all two-way joins, and an indexing structure to compute n-way joins for n>2. The attribute A0 is the TID. The TBS is stored by TID.TS value and the ABS by attribute value. The two-way joins can be found by taking the Cartesian product of the TID.TSs from the corresponding alpha-relations in the ABS. Figure 3 shows an index thread for computing a 4-way equijoin, starting at the arrow pointing to TID.TS 16. It is assumed that the alpha-relations shown in Fig. 3 have already been filtered by any select operations that may have been present in the corresponding production. By prefiltering selects and precomputing two-way joins, the architecture supports condition membership, memory support, and condition relationship. A more detailed description of the method for distributing the database and performing joins on this architecture can be found in [Brant87].

![Figure 3](image-url)
5. Current Research

Current work is focusing on the definition of the language, an analysis of the tradeoffs involved in using rule-level parallelism, and an implementation of the system on the Symult 2010 parallel computer. The desired language will incorporate constructs which would allow the programmer to specify operations on entire relations. This would provide the lazy mechanism with the capability to produce more than one instantiation if that is the objective of the programmer. The Symult machine should be quite effective at computing these set oriented operations. The tradeoff analysis in specifying the details of the lazy match algorithm will be done through the analysis of several applications. Some initial results have been gathered on the efectiveness of lazy matching at pruning the search space. These are shown in Table 1. For the applications that were analyzed an average of 60% of the instantiations computed by RETE or TREAT are never fired. To help understand the effect of WM size on performance, two of these test applications, WALTZ and TOURNEY, are being scaled up in the number of WMEs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Instantiations</th>
<th>Rule Firings</th>
<th>Unused Instantiations</th>
<th>% Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>WALTZ</td>
<td>151</td>
<td>70</td>
<td>81</td>
<td>54</td>
</tr>
<tr>
<td>ROBOT</td>
<td>478</td>
<td>410</td>
<td>68</td>
<td>14</td>
</tr>
<tr>
<td>TOURNEY</td>
<td>2324</td>
<td>528</td>
<td>1796</td>
<td>77</td>
</tr>
<tr>
<td>JIG25</td>
<td>205</td>
<td>58</td>
<td>147</td>
<td>72</td>
</tr>
<tr>
<td>MESGEN</td>
<td>860</td>
<td>138</td>
<td>722</td>
<td>84</td>
</tr>
</tbody>
</table>

Table 1.

6. Conclusion

An I/O Engine architecture which integrates the functionality of both a database and an expert system has been described. A key feature to the overall system is the introduction of a lazy matching algorithm with a linear worst case space complexity with respect to the number of elements in the database (working memory). This is a significant improvement over existing production systems having an exponential worst case space complexity. The data storage model for the database incorporates many eager matching techniques such as condition membership, memory support, and a new form of condition relationship through
the use of the attribute based schema. These techniques, taken as a whole, show great promise for the realization of an expert database system.
REFERENCES


