Effects of Database Size on Rule System Performance: Five Case Studies

David A. Brant[†], Timothy Grose[†], Bernie Lofaso[†], & Daniel P. Miranker[‡]

[†]Applied Research Laboratories
The University of Texas at Austin
P.O. Box 8029
Austin, TX 78713-8029

[‡]Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

Building practical expert database systems requires an effective inferencing capability over large data sets. Inferencing in this context means repeatedly executing a fixed set of queries, interleaved with update transactions, until a fixed point is reached. The effectiveness of the inferencing mechanism is heavily dependent upon the amount of state space needed and the ability of the underlying algorithms to avoid unnecessary work. Common techniques used in the design of rule-based systems store large amounts of state in order to derive precise query support information that will enable better performance. These techniques were intended for use in main memory on small data sets and are not necessarily suited for a database environment. When confronted with a large database these techniques may experience severe performance problems - severe enough to render them useless. In this paper we examine the effects of database size on five test cases. The use of real programs with real data provides insights that are not to be found through analysis and simulation. We compare two different rule systems, one based on the TREAT match algorithm and the other on LEAPS, a lazy matching algorithm. The results show that state can be a problem in rule systems and that by using lazy matching it is possible to eliminate some state while improving performance.

1 Introduction

A practical integrated database/rule system will require an efficient inferencing capability on large databases. Our research focuses on active forward chaining inferencing systems and their behavior in a database environment. Many studies have been done on various aspects of integrating rule and database systems. Several have dealt with performance issues [Tan, Maheshwari, & Srivastava, 1990, Cohen, 1989, Wang, 1989, Delcambre & Etheredge, 1988, Bein, King, & Kamel, 1987]. Most of these are analytical in nature and/or simulate synthesized rules and data. While analytical and simulated approaches to this problem are valuable, important issues such as realistic query mixes, rule execution order, and join selectivity factors, are heavily influenced by both the rule program itself and the data on which it operates. Therefore, we have assembled a suite of five real world programs with automatically extensible databases for use in understanding rule-based system behavior as the size of the database grows[1]. This work is being done in the context of developing DATEX, a tightly integrated expert database system.

In the interest of providing effective integration of rule execution with database systems, the behavior of incremental match techniques, first developed for main-memory AI systems, is investigated. These techniques take advantage of a space-time trade-off and store large amounts of state in order to derive precise query support information. We explore the utility of TREAT match [Miranker, 1990] and compare it against LEAPS [Miranker & Brant, 1990], a slightly more complicated algorithm, with better space characteristics, intended to support inferencing on databases.

In effect, rule-based systems execute a set of queries, interleaved with updates, in a cycle until reaching a fixed point. In a naive implementation, this can easily imply executing 10^2 - 10^3 multiway joins each cycle for thousands of cycles. In practice, that number is significantly smaller. This is due to the key observation that these systems display *temporal redundancy*, i.e., from one cycle to the next, there is very little change to the database. Thus, if knowl-

This work was supported by the ARL:UT Internal Research and Development Program.

1. This suite is available from the authors and contributions of new programs and data generators are welcome.

edge about the system state is retained across cycles, the amount of work done on any one cycle can become quite manageable by just incrementally updating that state. In the AI community the use of incremental match algorithms forms the basis of effective main-memory rule execution environments [Forgy, 1982]. The database problem of maintaining a materialized view is an incremental match problem without negation. A large number of incremental match algorithms have been described. Many of the newer ones have been developed for database applications [Wang, 1990, Blakeley & Martin, 1990, Srivastava, Hwang, & Tan, 1990, Raschid, Sellis, & Lin, 1988].

The problem with incremental matching is that the state for most algorithms has an exponential worst-case space requirement. For t tuples in a database and a rule with j joins, the worst-case state is $O(t^j)$. While real systems don't experience the worst case, any unexpected large polynomial space would still present significant concerns. To explore the possibility that programs might display this type of bad behavior, we conducted a scaling study in which the databases of five real programs were scaled up in size as their performance characteristics were measured. The goal was to escape the temptation to use arbitrary and manufactured values of selectivity for the select and join operators of rule systems, and to provide for the interaction of many rules acting in concert. Achieving this level of realism through analysis and simulation is extremely difficult, if not impossible.

The programs in the test suite are written in OPS5 [Forgy, 1981]. While OPS5 may be widely regarded as antiquated, it nonetheless embodies many of the underlying AI techniques used in other, more powerful, languages. Moreover, since OPS5 has been around for such a long time there exist public domain systems and numerous programs. Our empirical data was gathered using two of the best performing OPS5-based systems in existence. Both are compiled systems. The first uses a TREAT match algorithm which has been shown to be superior to the more widely used Rete match for most programs. The second uses a relatively new technique, called LEAPS, based on the idea of *lazy matching*.

Section 2 presents a brief description of rule-based systems, the four knowledge types that can be employed by their incremental match algorithms, and an overview of some of the published algorithms. In section 3 we describe the metrics used in the study and the test cases. Section 4 presents our empirical results. Section 5 contains a summary and our concluding remarks.

2 Rule-Based Systems

A rule-based program is a set of rules of the form if $P_1 \wedge ... \wedge P_n$ then $A_1,...,A_m$

where $P_1 \wedge ... \wedge P_n$ is a conjunction of n conditions or predicates on the current state of the base relations of the database, and $A_1,...,A_m$ is a set of m actions or update transactions on the database. In a relational model, each P_i is a predicate on a single relation. These predicates may contain constants and variables. Predicates containing constants can be partially evaluated with the relational *select* operator. If predicates share a variable, it must be consistently bound between them for the predicates to be satisfied. This is accomplished by a relational *join* operator (including both equijoin and non-equijoin). Execution of the program proceeds by evaluating the predicates, choosing one rule whose predicates are satisfied, executing its actions, and repeating the cycle until a fixed point is reached.

2.1 Query Support and Match Algorithms

Each unique set of tuples that satisfies a rule's predicates, $P_1 \wedge ... \wedge P_n$, is an *instantiation*. Thus, the set of instantiations at any one time is composed of the union of the temporary relations resulting from the queries associated with those predicates. A critical component of any rule-based system is the match algorithm that computes instantiations. A naive algorithm for finding the instantiations of the rules would execute the query associated with each rule's predicates against the entire database on each cycle. That approach is combinatorially explosive and computationally intractable. To address this problem, rule-based systems maintain state information from cycle to cycle. This information provides query support for the match algorithm. The knowledge provided by the state information has been divided into four categories [McDermott, Newell, and Moore, 1978, and Miranker, 1990].

1. *Condition Membership:* Associated with each P_i is a running count of the number of tuples that satisfy its select operation. This information is used to identify rules that are *active*, i.e., those with a non-zero count for each P_i. *Inactive* rules may be ignored by the match algorithm.
2. *Memory Support:* Provides explicit knowledge about precisely which tuples satisfy the selects of which P_i. These have been referred to as α-*memory*. If a separate set of tuples is maintained for each P_i, duplicate information will be stored for those predicates that have the same select criteria on the same relation. Those systems that eliminate this redundant data are said to use *shared* α-*memory*.
3. *Condition Relationship:* Provides knowledge about the interaction of predicates, i.e., the intermediate results of multiway joins are explicitly stored from cycle to cycle. These have been referred to as β-*memory*.
4. *Conflict Set Support:* the *conflict set* containing all of the instantiations for all rules is retained from cycle to cycle.

We have added a fifth knowledge type based on a lazy approach to query evaluation. We call the new type *Reso-*

Table 1 Relative Cost/Benefit Ranking of Rule Knowledge

Knowledge Type	Space-Time Cost	Expected Benefit
Condition Relationship (CR)	very high	very low
Memory Support (MS)	high	high
Condition Membership (CM)	low	high
Conflict Set Support (CS)	?	?
Resolution Support (RS)	?	?

Table 2 Match Algorithms and Their State Information

Algorithm	CR	MS	CS	CM	RS
Rete [Forgy, 1982]	✓	✓			
TREAT [Miranker, 1990]		✓	✓	✓	
Lazy [Miranker & Brant, 1990]		✓		✓	✓
Matchbox [Perlin, 1989]	✓	✓			
Gridmatch [Tan, Maheshwari, & Srivastava, 1990]	✓	✓			
unnamed [Rachid, Sellis, & Lin, 1988]	✓	✓			
Rime [Hwang, 1989]	✓	✓			
unnamed [Oflazer, 1986]	✓	✓			

lution Support. An underlying theme in current match algorithms is the eager evaluation of all active rules to generate all possible instantiations. However, on any given cycle only one instantiation is chosen to fire one rule. The choice is based on a conflict set resolution strategy. This observation has led to the development of LEAPS, a lazy matching algorithm that computes a fireable instantiation by using the conflict set resolution strategy to direct a best-first search. In doing so, it eliminates the conflict set and replaces it with a stack containing the state information needed to control a demand driven stream-based query process. Thus, key issues include the effects lazy matching has on performance and its space requirements compared to eager schemes.

Table 1 shows the knowledge techniques with a summary of their relative cost and expected benefit. As can be seen, the merits of using conflict set support versus resolution support are left as an open question. We will attempt to resolve these question marks based on the case studies in the following sections.

The values that are given in Table 1 are based on several observations. Since condition relationship stores the results of all intermediate joins it is given a very high space-time cost. Furthermore, TREAT, which does not use this knowledge, consistently outperforms match algorithms which do. Therefore, its expected benefit is stated as very low.

Memory support stores all of the intermediate results of select operations. Since there is a one to many mapping from the database to α-memory, the cost is generally high for this knowledge. However, when the selectivity of the constant tests is reasonably low (as it is for most systems measured), the expected benefit in increasing the speed of joins is also high.

Condition membership is low in space cost since it only requires a few bytes to implement. If memory support is implemented, then its time cost is also extremely low, since all of the select operations will have to be performed anyhow. If memory support is not used then its time cost might be more accurately stated as moderate, as it will then have to do its own select operations.

Table 2 lists some of the match algorithms in the literature and their associated knowledge types. While not obvious, the selection of some knowledge types makes others of little use. For instance, if condition relationship is chosen, then condition membership may provide little or no benefit. Therefore, it is probably not reasonable to expect any one algorithm to use all of them.

2.2 Match Algorithms

Most rule systems in use today are based on some form of the Rete match. The two exceptions that we are aware of are TREAT and LEAPS. The following sections describe these at a high level.

2.2.1 Rete

The Rete match incorporates memory support and condition relationship. It compiles the queries of the rules into a discrimination network in the form of an augmented dataflow network. The input portion of the Rete network contains chains of tests that perform the relational select operations. Tokens passing through those chains partially match a particular predicate and are stored in α-memory nodes, thus forming the memory support part of the algorithm. Following the α-memories are two-input nodes that test for consistent variable bindings between predicates. By analogy, these nodes incrementally compute the join of the memories on the input arcs. When a token enters a two-input node it is compared against tokens in the memory of the opposite arc. Paired tokens with consistent variable bindings are stored at the output of the two-input nodes as β-memories. Tokens that propagate from the last

β-memories in the network reflect changes to the conflict set.

The Rete algorithm is the basis for most rule-based systems in use today.

2.2.2 TREAT

It has been shown that the condition relationship information contained in the β-memory is not justified from a performance standpoint in main-memory systems [Miranker, Lofaso, Farmer, Chandra, & Brant, 1990] and this result has been recently extended to databases as well [Wang & Hanson, 1990]. The TREAT match algorithm does not use β-memories and has been shown to consistently outperform Rete-based systems. The rationale for this is due to the effects of the removal of tuples from the system. In general, Rete must do as much work to delete a tuple as it does to add one. TREAT does much less work on removals since it does not have to update the intermediate join results. TREAT does do slightly more work than Rete when tuples are added, but for most programs, the trade-off favors TREAT.

From a database perspective, obviating the need for maintaining intermediate join results is crucial, since they constitute the largest consumer of memory in Rete-based systems. However, TREAT still makes use of three categories of state information - condition membership, memory support, and conflict set support. It is interesting to note that TREAT represents a case of reducing space requirements while at the same time improving performance.

2.2.3 LEAPS

An underlying theme in both Rete and TREAT is the eager evaluation of all active rules to generate all possible instantiations. However, on any given cycle only one instantiation is used to fire one rule. This observation has led to the development of the lazy matching algorithm known as LEAPS. LEAPS computes at most one instantiation on any given cycle. This is done by setting up demand driven data streams used to produce instantiations. In doing so, it eliminates the conflict set and replaces it with a stack. The stack contains information needed to restart streams that have been temporarily suspended due to the processing of a higher priority stream. One of the issues to be examined in the study is how big of a space problem is the conflict set and is the stack an improvement.

3 The Test Suite and Metrics

3.1 The Programs and Data Generators

Each of the selected programs was chosen for a specific reason. The goal was to achieve a diversity in terms of the number of rules, the AI problem solving technique employed, and the problem domain. First, it was necessary to be able to generate randomly large databases for the programs to work on. It was also desirable that these databases be automatically produced under controlled parameters. Second, we wanted a set of programs that varied considerably in size and complexity. The resulting suite contains programs that range from several rules to over seven hundred rules. And, third, we wanted to chose a set of programs that spanned the range of standard expert system problem solving techniques, e.g., constraint propagation, blackboards, and A* search. Table 3 provides a summary of the major characteristics of the test suite.

Table 3 Program Summaries

Name	No. of Rules	Comment
manners	8	Finds a seating arrangement for dinner guests by depth-first search.
waltz	33	Waltz line labeling for simple scenes by constraint propagation.
waltzdb	35	A more database oriented version of Waltz line labeling for complex scenes by constraint propagation.
ARP	118	Route planner for a robotic air vehicle using A*.
weaver	637	A VLSI router using a blackboard technique.

3.1.1 Manners

Manners was derived from a program appearing in Kiernan, de-Maindriville, & Simon, 1990. It is based on a depth-first search solution to the problem of finding an acceptable seating arrangement for guests at a dinner party. The particular seating protocol enforced by the version used in this study ensures that each guest is seated next to someone of the opposite sex who shares at least one hobby. The manners program can be extended quite easily to handle many different criteria for constraining the seating arrangement. Further it is very easy to adjust the join selectivity of the program by controlling the distribution of the hobbies. The data used in this study gave a uniform distribution of hobbies from a minimum of 2 to a maximum of 5 per guest. Guests were evenly divided into male and female.

3.1.2 Waltz and Waltzdb

Waltz was developed at Columbia University. It is an expert system designed to aid in the 3-dimensional interpretation of a 2-dimensional line drawing. It does so by labeling all lines in the scene based on constraint propagation. Only scenes containing junctions composed of two and three lines are permitted. The knowledge that Waltz uses is embedded in the rules. The constraint propagation consists of 17 rules that irrevocably assign labels to lines based on the labels that already exist. Additionally, there are 4 rules that establish the initial labels. The rules provide the explicit constraint information by means of constant tests - there is no generalized form of constraint

propagation. The significance of this became apparent when we tried to expand the Waltz program to handle the more general case of line drawings involving junctions composed of 4, 5, and 6 lines. This resulting program is called waltzdb.

Waltzdb was developed at the University of Texas at Austin. It is more general version of the Waltz program described in the previous section. Waltzdb is designed so that it can be easily adapted to support junctions of 4, 5, and 6 lines. The method used in solving the labeling problem is a version of the algorithm described by Winston [Winston, 1984]. The key difference between the problem solving technique used in waltz and waltzdb is that waltzdb uses a database of legal line labels that are applied to the junctions in a constrained manner. In Waltz the constraints are enforced by constant tests within the rules.

The input data for waltz is a set of lines defined by Cartesian coordinate pairs. The data generator uses a base drawing consisting of 72 lines which we refer to as a *region*. The user can specify any number of regions to be generated. We have run test cases with as many as 100 regions.

3.1.3 ARP

The Aeronautical Route Planner calculates the lowest cost route between two points for an airplane or missile. It calculates the route based on terrain, threat, and cost data. The route will avoid terrain and surface-to-air missile sites (the threats). ARP minimizes three costs for a route: the cost of travelling a distance, the cost of being at an altitude, and the cost of being at a height above the terrain. ARP uses the A* search algorithm to restrict the portion of the search space it needs to examine to calculate the route. The input data is a database of terrain information for a given corridor.

3.1.4 Weaver

Weaver is an expert system designed to perform VLSI channel and box routing [Joobbani & Siewiorek, 1986]. It is a large complex program that is made up of several independent expert systems communicating via a common blackboard. Its input data is a list of pins, nets, and channel dimensions.

3.2 Metrics

In order to minimize the affects of constants we have chosen to represent all of the memory usage in terms of the number of *data elements*. Data elements for both TREAT and LEAPS include the tuples of the database and α-memory. TREAT also contains entries in the conflict set, while LEAPS keeps a stack.

There are two primary measures of time used in this study. The first is the overall execution time of the programs in terms of the number of cpu seconds. Since match time dominates the overall execution time for these systems, the second measure is the number of times an α-memory is touched in the joins of the matching algorithm. This is further broken down into successful versus unsuccessful tests. Successful tests are important in that they indicate the minimum work necessary given a perfect indexing method on the join attributes.

4 Results

Each program was run with four equally spaced sets of data points (shown in Table 4). Table 5 shows the counts

Table 4 Initial Database Size

Program	Set 1	Set 2	Set 3	Set 4
manners	16	32	64	128
waltz	864	1800	2664	3600
waltzdb	288	576	864	1152
ARP[1]	106	106	106	106
weaver	531	791	1311	1831

of the measured elements at the point when they were at their maximum. There are several things to notice in this data. First, the stack is very small and remains that way as database size increases. In fact, for four of the five programs it is a small constant. Contrast this to the size of the conflict set and its growth. Figure 1 graphs this data for all five programs. Also note that we have identified non-linear behavior for the conflict set in the manners program.

Another interesting aspect of these programs was the size of the α-memory. Both of the systems used to gather the data do not implement sharing of α-memory, but clearly the dominant space factor is α-memory. We compared the α-memory size to the maximum size of the database during the program execution and discovered that the replication factor of data in the α-memory was much higher than anticipated. Table 6 compares the maximum size of the database with the measured maximum α-memory size (columns 2 and 3). Based on this data we calculated the average number of α-memory entries required for each database entry. Since any change to the database requires changes to the α-memory, this can also be interpreted as an update ratio (column 4).

In order to analyze the effects of using shared α-memory, we used traces of the executions to drive a shared α-memory simulator. The result is shown in column 5. By dividing the measured data in column 3 with the simulated data in column 5 we derived the average sharing ratio, i.e., the number of α-memory entries that are subjected to the same select operator. The result was surprising and demonstrated the clear benefits of using sharing. The last column shows the expected α-memory updates per database change based on sharing.

Table 7 shows the total execution time for the test programs. The data is graphed in Fig. 2. To get these times the programs were run with uninstrumented versions of the

Table 5 Maximum Counts

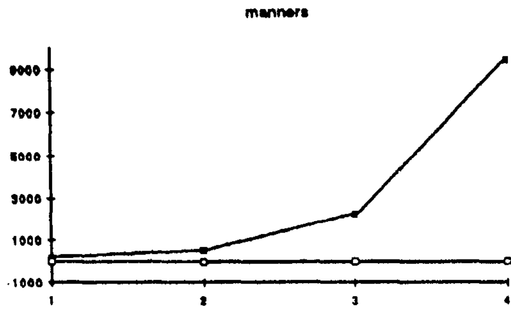
Program	Element		Data Set 1	Data Set 2	Data Set 3	Data Set 4
manners						
	TREAT:	database	231	702	2,425	8,952
		α-memory	797	2,570	9,227	34,856
		conflict set	174	552	2,248	9,490
	LEAPS:	database	231	702	2,425	8,952
		α-memory	797	2,570	9,227	34,856
		stack	1	1	1	1
waltz						
	TREAT:	database	2,749	5,505	8,049	10,805
		α-memory	98,403	197,203	288,403	387,203
		conflict set	3,192	6,416	9,392	12,616
	LEAPS:	database	2,749	5,505	8,049	10,805
		α-memory	98,403	197,203	288,403	387,203
		stack	4	4	4	4
waltzdb						
	TREAT:	database	3,272	5,864	8,456	11,048
		α-memory	124,056	222,296	320,536	418,776
		conflict set	1,208	2,200	3,192	4,184
	LEAPS:	database	3,200	5,736	8,272	10,808
		α-memory	120,431	215,815	311,199	406,583
		stack	2	2	2	2
weaver						
	TREAT:	database	575	815	1,335	1,855
		α-memory	150,073	212,632	348,552	489,232
		conflict set	35	359	1,079	1,799
	LEAPS:	database	575	815	1,335	1,855
		α-memory	143,128	207,872	348,552	489,232
		stack	2	2	2	2
arp						
	TREAT:	database	494	747	920	1,116
		α-memory	6,439	10,023	12,440	15,214
		conflict set	353	601	796	960
	LEAPS:	database	494	747	920	1,116
		α-memory	6,447	10,031	12,494	15,278
		stack	15	20	25	30

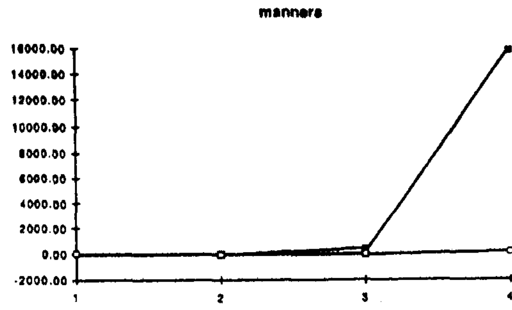
Table 6 Effects of Using Shared α-Memory

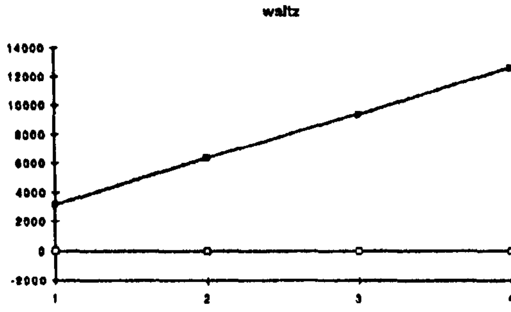
Program	Database	α-Memory	Update Ratio	Shared α-Memory	Sharing Ratio	Update Ratio (S)
manners	1,855	489,232	263.7	12,620	39.8	6.8
waltz	8,952	34,856	3.9	9,146	3.8	1.0
waltzdb	10,805	387,203	35.8	53,205	7.3	4.9
weaver	11,048	406,583	36.8	19,687	20.6	1.8
arp	1,116	15,278	13.7	13,196	1.2	11.8

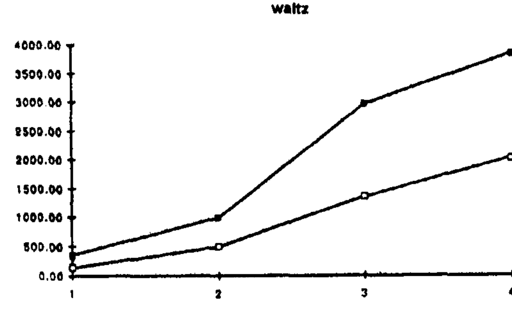
Table 7 Total Execution Time (in seconds)

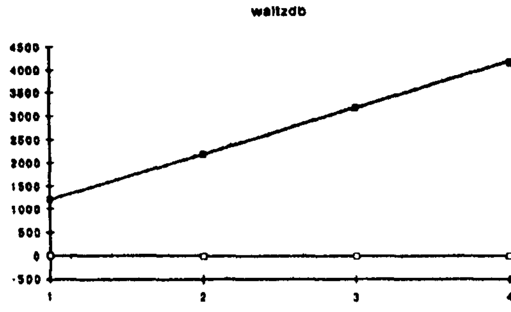
Program	Match Alg.	Set 1	Set 2	Set 3	Set 4
manners	TREAT	1.0	13.8	425.8	15,838.5
	LEAPS	0.3	1.0	10.4	153.1
waltz	TREAT	343.3	988.0	2,963.0	3,831.8
	LEAPS	147.9	502.1	1,353.5	2,009.6
waltzdb	TREAT	641.5	2,109.6	4,341.6	8,033.3
	LEAPS	102.9	340.0	796.8	1,298.3
weaver	TREAT	170.3	255.8	552.6	1,053.7
	LEAPS	138.5	232.4	416.6	680.4
arp	TREAT	224.3	529.8	822.1	1,220.2
	LEAPS	96.9	224.6	325.0	464.2

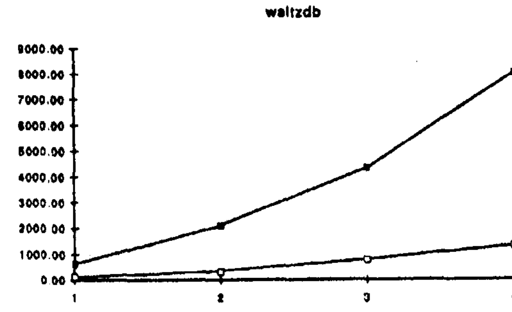


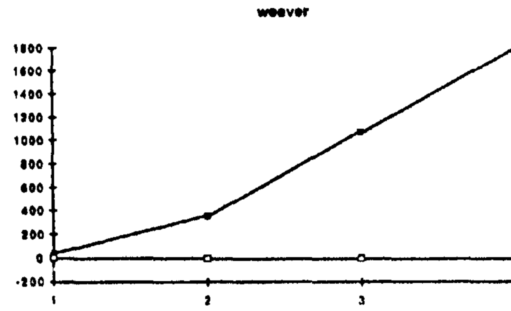


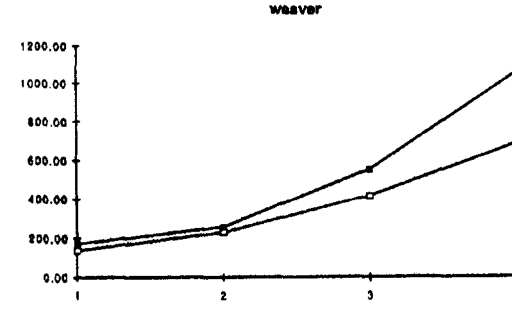












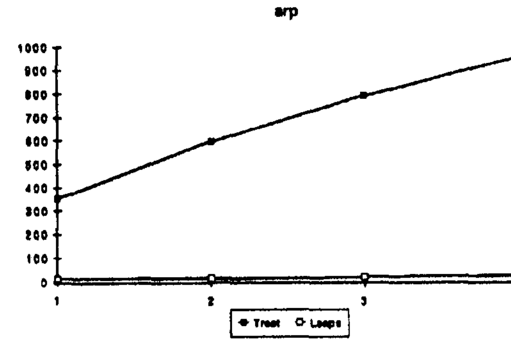


Figure 1: Conflict Set vs Stack

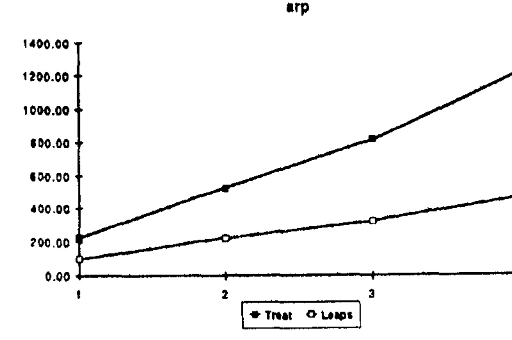


Figure 2: Execution Time

Table 8 α-Memory Tests

Program/Match Alg.		Set 1	Set 2	Set 3	Set 4
manners					
TREAT:	Success	194,806	4,246,944	126,836,779	4,139,524,711
	Fail	43,899	365,976	4,073,813	50,165,431
	Total	238,705	4,612,920	130,910,592	4,189,690,142
LEAPS:	Success	9,996	142,551	2,180,071	34,180,273
	Fail	4,001	26,226	192,883	1,471,733
	Total	13,997	168,777	2,372,954	35,652,006
waltz					
TREAT:	Success	5,772,411	23,048,007	49,204,311	88,600,707
	Fail	41,630,145	167,994,682	359,850,007	649,172,207
	Total	47,402,556	191,042,689	409,054,381	737,772,914
LEAPS:	Success	842,120	3,312,354	7,037,370	12,638,004
	Fail	23,222,522	93,603,118	200,425,822	361,490,547
	Total	24,064,642	96,915,472	207,463,192	374,128,551
waltzdb					
TREAT:	Success	64,310,313	204,316,661	423,272,961	721,179,213
	Fail	83,128,182	263,469,974	545,241,334	928,442,262
	Total	147,438,495	467,786,635	968,514,295	1,649,621,475
LEAPS:	Success	10,984,240	35,885,166	75,111,356	128,662,810
	Fail	10,565,527	34,285,055	71,568,423	122,415,631
	Total	21,549,767	70,170,221	146,679,779	251,078,441
weaver					
TREAT:	Success	12,837,799	19,333,608	37,564,688	62,782,968
	Fail	12,333,757	18,384,427	41,594,167	89,466,307
	Total	25,171,556	37,718,035	79,158,855	152,249,275
LEAPS:	Success	10,295,048	16,192,185	32,105,485	53,921,185
	Fail	8,934,632	13,317,283	25,535,823	43,426,363
	Total	19,229,680	29,509,468	57,641,308	97,347,548
arp					
TREAT:	Success	12,471,770	32,094,819	51,553,953	79,816,302
	Fail	7,011,513	22,319,887	39,214,120	64,792,582
	Total	19,483,283	54,414,706	90,768,073	144,653,884
LEAPS:	Success	5,166,769	11,688,013	17,777,300	25,781,141
	Fail	2,765,489	8,819,169	15,541,914	25,715,333
	Total	7,932,258	20,507,182	33,319,214	51,496,474

compilers that do not gather statistics. Therefore, the times provide a good indication of the state of the art in main memory rule-based system execution. The largest database size in the applications was 11,048 for waltzdb. Its program took 1,298.3 seconds to complete. All of the timing data was derived from runs on Sun SPARCstation 1+ workstations with 64MBytes of main memory and local disks.

Most of the time spent in these systems goes into the join work. Joins are performed on the α-memory. Table 8 shows how many times an α-memory element is touched in the life of the program. The systems we used do not provide any attribute value based indexing on α-memory. For large scale database problems this should prove useful. To gain some insight into how useful it might be, we counted both successful and unsuccessful tests (test of an α-memory element is a test of its value on the join attribute). The successful tests provide a measure of the minimum amount of work that must be done by the joins. By dividing the successful tests by the total tests we can also arrive at an overall join selectivity for the programs. Table 7 shows the join selectivity data. We were somewhat surprised by the high selectivity. However, these databases are set up to provide specific knowledge to the expert system. This may not be the case at all when an expert system is used on a general purpose database. In that situation, join selectivity may be considerably lower.

5 Conclusion

Several clear observations are possible based on the data gathered. First, α-memory that is not shared may lead to excessive space requirements and update costs. Second, the conflict set for some applications does become problematic as the database grows, in that it exhibits a non-linear space behavior. Third, the stack size in LEAPS remains extremely low and constant for most applications. And, fourth, the lazy matching strategy of LEAPS provides significant speedup that improves as the problem size increases.

Table 9 Average Composite Join Selectivity

Program	Match Alg.	Set 1	Set 2	Set 3	Set 4
manners	TREAT	0.82	0.92	0.97	0.99
	LEAPS	0.71	0.84	0.92	0.96
waltz	TREAT	0.12	0.12	0.12	0.12
	LEAPS	0.03	0.03	0.03	0.03
waltzdb	TREAT	0.44	0.44	0.44	0.44
	LEAPS	0.51	0.51	0.51	0.51
weaver	TREAT	0.51	0.51	0.47	0.41
	LEAPS	0.54	0.55	0.56	0.55
arp	TREAT	0.64	0.59	0.57	0.55
	LEAPS	0.65	0.57	0.53	0.50

Table 10 Relative Cost/Benefit Ranking of Rule Knowledge

Knowledge Type	Space-Time Cost	Expected Benefit
Condition Relationship (CR)	very high	very low
Memory Support (MS)	high	high
Condition Membership (CM)	low	high
Conflict Set Support (CS)	**high**	**moderate**
Resolution Support (RS)	**low**	**high**

Based on these results we can complete the cost/benefit matrix from Table 1. Recall that the values for conflict set support and resolution support were left as question marks. Table 10 shows the completed comparison. The high cost of conflict set support is derived from a comparison of its space and effects on execution time to the LEAPS-based approach. Since conflict set support does enable TREAT to execute faster than Rete-based systems, its benefit is rated as moderate. From the table we conclude that two of the knowledge types, condition membership and resolution support, are clearly desired for an integrated database/rule system. These two obviate the need for condition relationship and conflict set support. Therefore, we are now turning our attentions to the last one — memory support. We feel certain that a more effective replacement for memory support can be developed. The key will be to find ways of providing the information contained in the α-memory without relying on the accompanying space. Both TREAT and LEAPS are examples of this reasoning successfully applied to the problem of integrating rule systems with databases.

6 References

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

Blakeley, J.A. and N.L. Martin, "Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis", *Proceeding of the Sixth International Conference on Data Engineering*, pp. 256-263, February, 1990.

Cohen, D., "Compiling Complex Database Transition Triggers," *Proc. ACM SIGMOD Conference*, pp. 225-234, Portland, Oregon, May, 1989.

Delcambre, L.M.L. and Etheredge, J.N., "The Relational Production Language: A Production Language for Relational Databases", *Proceedings of the Second International Conference on Expert Database Systems*, 1988.

Forgy, C., "OPS5 User's Manual", *Technical Report CMU-CS-81-135*, Carnegie-Mellon University, 1981.

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

Joobbani, R., and D.P. Siewiorek, "WEAVER: A Knowledge-Based Routing Expert," *IEEE Design and Test of Computers*, pp. 12-23, Feb., 1986.

Kiernan, G. , C. de-Maindriville and E. Simon, "Making Deductive Database a Practical Technology: A Step Forward," *Report No. 1153*, Institute National de Recherche en Informatique et en Automatique, January, 1990.

McDermott J., A. Newell, and J. Moore, "The Efficiency of Certain Production System Implementations," In *Pattern-directed Inference Systems*, D. Waterman and F. Hayes-Roth (eds.), Academic Press, 1978.

Miranker, D., *TREAT: A New and Efficient Match Algorithm for AI Production Systems*, Pittman/Morgan Kaufman, 1990.

Miranker, D., B.J. Lofaso, G. Farmer, A. Chandra, and D. Brant, "On a TREAT Based Production System Compiler," *Proceedings of the 10th International Conference on Expert Systems*, Avignon, France, 1990.

Miranker, D.P., and D.A. Brant, "An Algorithmic Basis for Integrating Production Systems and Database Systems," *Proceedings Sixth International Conference on Data Engineering*, February, 1990.

Oflazer, K., "Partitioning and Parallel Processing of Production Systems," Dept. of Computer Science, Carnegie-Mellon University, 1986.

Perlin, M.W., "The Match Box Algorithm for Parallel Production System Match," Department of Computer Science, Carnegie Mellon University, CMU-CS-89-163, May, 1989.

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.

Srivastava, J., K-W Hwang and J. S. Tan, "Parallelism in Database Production Systems," *Proceeding Sixth International Conference on Data Engineering*, pp. 121-128, 1990.

Tan, J.S.Eddy, M. Maheshwari, and J. Srivastava, "GridMatch: A Basis for Integrating Production Systems with Relational Databases," *CS TR 90-14*, Computer Science Dept. Univ. of Minnesota, 1990.

Wang, C. K., "Rime: A Match Algorithm for MRL," Dept. of Computer Sciences, The University of Texas at Austin, unpublished manuscript, November, 1989.

Wang, Y-W, and E. Hanson, "A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions," Wright State University, Technical Report WSU-CS-90-18, 1990.

Winston, P., *Artificial Intelligence*, Adison-Wesley Publishing Co., Reading, Mass., 1984.