PATTERN MATCH REDUCTION FOR THE RELATIONAL PRODUCTION LANGUAGE IN THE USL MMDBS

Lois M.L. Delcambre†, Jint Waramahaputi†, and James N. Etheredge‡

†The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, Louisiana 70504
(318) 231-5697

‡Computer Science Department
California Polytechnic State University
San Luis Obispo, California 93407
(805) 756-2921

Abstract

The Relational Production Language, RPL, is a rule language where both the condition and action part of a rule is expressed in a relational language. This makes RPL a natural choice for expert database systems. In this paper, the performance of the RPL interpreter is addressed through the use of Pattern Match Reduction, PMR, within the context of the USL main memory database system, the USL MMDBS. PMR provides a generalization of relational view materialization by supporting relationally complete queries and by allowing the materialization of any/all intermediate nodes in a query tree. The USL MMDBS supports PMR because of its performance characteristics for query processing (due to a linear sort/join algorithm) and because intermediate query results are stored as "first class" relations. Thus the materialization required by PMR is directly supported. This paper introduces PMR and the USL MMDBS and presents preliminary performance results.

1. Introduction

An expert database system (EDS) is a technology that intends to capture the benefits of both a rule-base expert system (ES) and a conventional database management system (DBMS). Although there is a lot of interest in the development of an EDS, there is some debate concerning the appropriate language and underlying formal model. Some researchers have embraced an ES language like OPS5 and considered an implementation against a disk-resident DBMS [Sellis88]. Others have tried a similar approach using Prolog [Ioannidis88, Napheys88]. Within the database arena, extensions to a relational query language using database procedures have been studied [Stonebraker87, Hanson88]. Finally, the use of Datalog and its extensions have been widely studied by the deductive database community, e.g. [Ullman88].

This research adopts the Relational Production Language, RPL, as an appropriate language for EDSs. RPL rules, also called relational rules, consist of a condition/action pair. The condition is expressed as a relational query that addresses working memory described as a relational database and each answer tuple comprises an instantiation for the rule. Once an instantiation for a particular rule is chosen for firing by the conflict resolution process, then the action portion of the rule updates working memory within the context established by the rule condition. In other words, the action part of the rule can modify or delete objects that participate in the instantiation. This language was proposed [Delcambre88a, b, c] as a language for an EDS particularly for the case when the rule program must access existing, conventional databases.

RPL is also a very attractive alternative for deductive database rule programs. RPL is essentially similar to RDL/1 [Maindreville88a, b, Simon88], an independently developed language for deductive databases. RDL/1 has been shown to be strictly more expressive than Datalog, even with negation, because it allows multiple actions including deletions in the head of a rule. Through the integration of the RPL and RDL/1 research paths, the language is being promoted as a rule language for all database purposes [Delcambre89].

Clearly, RPL can be used as an expert database system language. One serious challenge presented by RPL concerns performance. Can RPL and the relational paradigm compete with a conventional rule language and the Rete algorithm? In this paper, this question is addressed in two main ways. First, the α and β memories normally associated with the Rete algorithm are replaced by materialized views for intermediate nodes of the

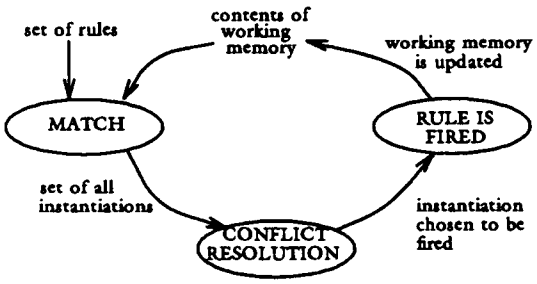
query tree that represents the condition portion of a relational rule. The technique is known as Pattern Match Reduction, PMR. The basic approach and preliminary performance results are presented here. Second, since expert system inferencing normally takes place in main memory, the use of a main memory database to support both the materialized views and the query processing component used to generate the conflict set is explored. The USL MMDBS supports a linear sort and represents ***all*** intermediate results in the same manner as base relations. Thus the USL MMDBS provides a linear join algorithm and also directly supports materialized views.

The paper is organized as follows. In Section 2, conventional expert system interpreters are briefly reviewed. In Section 3, the notion of Pattern Match Reduction (PMR) is presented as an alternative to the Rete algorithm. In Section 4, the USL MMDBS is introduced along with a discussion of how it facilitates the RPL/PMR environment. The paper concludes with a discussion of this work in Section 5.

2. Motivation

Most interpreters for a rule-based language implement the recognize/act cycle for the purpose of identifying instantiations, choosing a rule to fire, and then firing the rule, as shown in Figure 2.1. During each cycle, all possible instantiations for rules is determined by the ***match*** process in Figure 2.1. The ***conflict resolution*** process chooses the instantiation that will fire. Normally, only ***one*** instantiation is chosen. Finally, the selected rule is ***fired*** and working memory is updated according to the set of updates in the action part of the rule.

Based on the concept of temporal redundancy, rule-based interpreters assume that the contents of



The Recognize/Act Cycle
Figure 2.1

working memory changes rather slowly, thus the conflict set, i.e. the current set of instantiations, also does not change much from one cycle of the interpreter to the next. The Rete network [Forgy81, 82] and associated algorithms were designed to facilitate the computation of the conflict set during each interpreter cycle. The function of the Rete network is two-fold. First, the network filters out objects that do not satisfy the condition part of individual rules. Every set of objects that satisfy the condition of a rule are produced at the bottom of the Rete network as part of the conflict set. Second, the Rete network incorporates α memories to store objects that satisfy the single input nodes (selection criteria) of a condition and the β memories queue objects involved in two input nodes (either cross product or set difference.)

For the implementation of RPL, there are several reasons why the Rete algorithm is not appropriate. First, individual relational rules are more expressive than OPS5 [Delcambre88a]. Therefore RPL requires something more general than the Rete algorithm. Second, because RPL naturally supports both inter-rule and intra-rule parallelism, the principle of temporal redundancy is not necessarily applicable.

The most natural (and the most straightforward) implementation for RPL relies on a relational query processor to check the conditions and execute the actions. However, query processing technology presents two drawbacks when considered as a possible implementation of RPL. The first problem is that the data is assumed to be disk resident and thus the query processor assumes a different time/space tradeoff than in a main memory system. The second problem is that RPL requires incremental processing of queries when base relations change. In this paper, the performance challenge is addressed in the framework of the relational model in two main ways. First, the concept of the Rete network is simulated and generalized through the use of PMR. Second, the interpreter uses the USL Main Memory Database System, MMDBS, to support query processing. These two concepts are described in Sections 3 and 4, respectively.

3. Pattern Match Reduction

It is standard practice in a relational DBMS to define a user view (or virtual relation) as a query. User queries can then be processed through query modification. Another possibility for view

processing is when the view is computed and explicitly stored in the database. This has the advantage of supporting queries against the view as simple queries involving just one relation, the materialized view. The disadvantage is that the materialized view must be patched up whenever the underlying base relations change. Algorithms for maintaining materialized views have been studied in the context of relational databases, most recently by Tompa and Blakeley [Tompa88, Blakeley86]. They restrict their attention to select-project-join queries where only the final query answer is materialized.

In this paper, the use of materialized views is proposed as an alternative to the α and β memories of the Rete network. View materialization for RPL rule processing is different from conventional rule processing for the following reasons.

1. Any or all of the intermediate nodes in a query tree might be materialized rather than just the final query answer.
2. The base relations in the RPL environment may experience massive changes during one interpreter cycle. First, since the action part of a relational rule consists of a set of database update operations, multiple base relations may change during one cycle of the interpreter. Second, a single relation may experience both additions and deletions within one cycle. Third, the RPL potential for intra-rule and inter-rule parallelism allows for a *set* of tuples to be added to or deleted from individual relations during one cycle.

These differences serve as the motivation for PMR, a formally defined procedure to support materialized views. The PMR formulas are shown in Figure 3.1 and are proven correct in [Etheredge89]. PMR formulas are relational algebra expressions used to compute the new query answer when the underlying base relations have been updated. Similar formulas for a subset of the relational algebra operators were presented in [Tompa88, Blakeley86] in the context of materialized views and in [Bancilhon86] in the context of Datalog programs for deductive databases. Each formula presented in Figure 3.1 presents the most general case when all of the base relations, denoted R or R_1, and R_2, experience both additions and deletions, denoted, e.g. R_{add} and R_{del}, in one interpreter cycle.

R/0	PMR Formula
σ	$\sigma(R) \cup \sigma(R_{add}) - \sigma(R_{del})$
π	$\pi(R) \cup \pi(R_{add}) - \pi(R_{del})$
$\times$	$[R_1 \times R_2]$ $\cup [R_1 \times R_{2add}]$ $\cup [R_{1add} \times (R_2 \cup R_{2add})]$ $-[R_{1del} \times (R_2 \cup R_{2add})]$ $-[((R_1 \cup R_{1add}) - R_{1del}) \times R_{2del}]$
$\cup$	$[R_1 \cup R_2]$ $\cup [R_{2add} - R_1]$ $\cup [R_{1add} - (R_2 \cup R_{2add})]$ $-[R_{1del} - (R_2 \cup R_{2add})]$ $-[R_{2del} - ((R_1 \cup R_{1add}) - R_{1del})]$
$-$	$[R_1 - R_2]$ $- [R_1 \cap R_{2add}]$ $\cup [R_{1add} - (R_2 \cup R_{2add})]$ $- [R_{1del} - (R_2 \cup R_{2add})]$ $\cup [((R_1 \cup R_{1add}) - R_{1del}) \cap R_{2del}]$

PMR Algorithm Summary
Table 3.1

The PMR formulas all adhere to the same basic structure. Given that the previous materialization of the node is called A_{old} and the new materialization is called A_{new}, each PMR formula has the following form:

$$A_{new} = A_{old} \cup A_{add} - A_{del}$$

The PMR formula directly computes A_{new} by computing A_{old}, A_{add}, and A_{del} to be used as input by a subsequent node in the query tree. Thus the PMR formulas naturally suggest an algorithm for the maintenance of materialized views. At every node, R, R_{add}, and R_{del} (or similarly for R_1 and R_2) are received as input and the PMR algorithm computes A_{old}, A_{add}, and A_{del}. Note that each PMR formula can take advantage of the case when the corresponding node of the query tree has been materialized. A_{old} need not be computed when this node is materialized because it is precisely the stored materialization.

The PMR formulas have a natural simplification when the changes to the base relations are less comprehensive than those listed above. As an example, if only one input relation, e.g. R_1, is changed for a cross product operation then both R_{2add} and R_{2del} are empty. The expression then simplifies to:

$$[R_1 \times R_2] \cup [R_{1add} \times R_2] - [R_{1del} \times R_2]$$

As another example, if only additions occur during one cycle, the cross product formula simplifies to

$$[R_1 \times R_2] \cup [R_1 \times R_{2add}] \cup [R_{1add} \times (R_2 \cup R_{2add})].$$

In order to illustrate the PMR algorithm, consider the query tree presented in Figure 3.2.

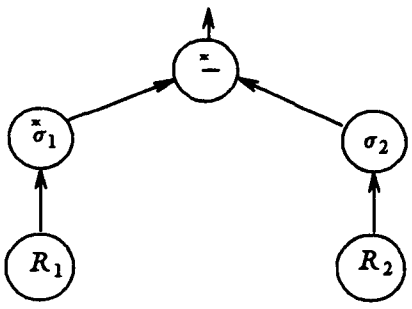
Assume that the σ_1 node and the set difference node are both materialized and denoted S and D respectively, and assume that tuples are added to both R_1 and R_2. The PMR algorithm proceeds as follows : (Steps 1 and 2 can proceed in parallel).

1. The σ_1 node is calculated as $S \cup \sigma_1 (R_{1add})$, denoted as R_3 and R_{3add}.
2. The σ_2 node is calculated as $\sigma_2 (R_2) \cup \sigma_2 (R_{2add})$, denoted R_4 and R_{4add}. Note that since σ_2 is not materialized, $\sigma_2 (R_2)$ must be recalculated.
3. The difference node is calculated as

   $$D \cup (R_{3add} - (R_4 \cup R_{4add}) - R_3 \cap R_{4add})$$

   In this case the materialization for D means that only the second and third terms must be calculated.

The PMR algorithms have been extensively simulated and compared to both the Rete algorithm and to a brute force query processing algorithm that uses no materialization [Etheredge89]. Rete is simulated (for select-join-difference queries) by materializing nodes that correspond to the α and β memories. Since a main memory query processing capability is assumed for RPL, there is *no* consideration for secondary storage. Also, this approach supports a direct comparison between Rete and PMR.



Sample Query Tree (* materialized node)
Figure 3.2

For the simulation, each node of the tree has a selectivity value that determines the size of the output relation. This figure was set to 25%, 50%, and 75% for various runs. The most appropriate use of PMR is shown to be in between the Rete algorithm and a brute force RPL algorithm. Preliminary conclusions from the PMR simulations are listed below.

1. For single tuple at a time processing with a relatively small number of changes to base relations, the Rete algorithm is superior. Thus, when the principle of temporal redundancy applies, Rete is a good choice.
2. For set oriented processing, RPL with PMR using a relational query processor performs significantly better than the Rete algorithm. This reinforces the idea that relational query processing is well-suited for set-oriented processing.
3. In cases where the selectivity is high (75% or greater) and the number of changes to base relations is high, then a brute force implementation of RPL (without PMR) provides the best performance. In the presence of massive changes, the complete recalculation of RPL is appropriate.
4. There are at least two motivations for materializing a node. First, when base relations do not change, a materialized node can completely avoid recalculating a subquery. Second, when base relation are updated, materialized nodes along the change path (between the changed base relation and the root of the query tree) facilitate PMR and tend to enhance performance.

Basically, the three approaches form an spectrum as follows:

Rete	base relations change slowly; one instantiation fired per cycle
RPL with PMR	moderate levels of change to base relations; sets of instantiations fired per cycle
RPL without PMR	massive changes to base relations; sets of instantiations fired per cycle

Note that the PMR algorithms were developed formally and were developed without regard for the actual main memory algorithms used to process relational algebra operators. Thus, the conclusions concerning RPL, PMR, and Rete are all made at a rather abstract level. In the next section, the USL MMDBS is presented and the support that it offers for RPL rule processing is

discussed in detail.

4. Main Memory Support for RPL

The performance requirements of an expert system demand that RPL relations be available in main memory for both RPL query processing and RPL update processing. The natural choice is to consider a main memory database system, MMDBS, for RPL rule processing. In this section, the use of an MMDBS to support PMR is discussed in Section 4.1. The design goals of the USL MMDBS are presented in Section 4.2, an overview of the USL MMDBS is given. Finally, time and space performance figures for query processing in the USL MMDBS are presented in Section 4.3.

4.1. Facilitating PMR in a MMDBS

The major performance challenge presented for any main memory query processing and storage facility is to provide fast processing while conserving space. In essence, a MMDBS must adjust to the new time/space tradeoff as compared with conventional database systems. Other differences between a DBMS and a MMDBS include the use of clustering. Traditional DBMS tend to cluster heterogeneous tuples that will be accessed concurrently in order to reduce seek time. This type of clustering is not an issue in main memory. On the other hand, contiguous storage of homogeneous data items (e.g. attributes values) can speed up both searching and scanning. This is particularly true for fixed length values because the address calculation is much faster.

In order to support RPL/PMR, there are additional performance issues to consider. Perhaps the most important concerns the storage of intermediate results of a query. Since PMR supports the materialization of *intermediate* nodes of a query tree, the query processor must access the materialized relations. One approach is to store intermediate relations in the same manner as base relations. Thus one of the design goals for the MMDBS is to provide efficient access to materialized nodes while minimizing the overhead associated with creating them. Note that in the PMR environment, it may be beneficial to dynamically adjust which nodes should be materialized while the rule program is running. In particular, the use of a self-controlling interpreter [Delcambre88b] lends itself to such dynamic control.

The RPL environment also presents an opportunity for performance improvement. Since the rule program is known to the interpreter (i.e. there is no need to support arbitrary, ad hoc queries), the storage structures can be optimized according to the rules. As an example, for a materialized select node, an index on the selection attribute will directly facilitate PMR and there is no need to worry about the other attributes. The only exception would be the case of materialized common subexpressions.

To summarize, the design goals for a MMDBS to support RPL/PMR are listed here.

1. Fast query processing for a relationally complete query language.
2. Efficient use of main memory.
3. Efficient support for sorting, searching and scanning, perhaps through the use of contiguous attribute values.
4. Appropriate support for intermediate relations to facilitate both query processing and PMR algorithms.
5. Minimal overhead to materialize an intermediate node, including the case when the decision to materialize a node is made dynamically.

4.2. Overview of the USL MMDBS

To accommodate the needs of RPL/PMR, a domain-based MMDBS called **USL** (yo**U** can **S**ort in **L**inear time) [Waramahaputi89] is presented. The USL MMDBS utilizes a simple, uniform database structure in order to reduce memory space required to store the database and provide fast, uniform access time for both relational algebra and database update on both permanent and temporary relations. The USL MMDBS relies primarily on an array structure and uses *no* indices. This eliminates both the storage required for the index and the overhead associated with updates. The basic idea is to maintain all domains explicitly and store them in sorted order. From the implementation perspective, this directly supports variable length attribute values, eliminates redundant representation of lengthy domain values, and represents the attribute with a relatively short ID (based on the number of domain values). From the perspective of the functionality supported, the domain representation of the USL MMDBS eliminates the need for encoding attributes. There is essentially no penalty for having a domain of {male, female}

as opposed to {m, f}. Thus, although the USL MMDBS was motivated by performance demands, the elimination of data encoding also provides a *friendly* user interface because the attribute values can be displayed to the user on a menu. Additionally, the USL MMDBS supports sophisticated queries such as "which relations contain information relating to color."

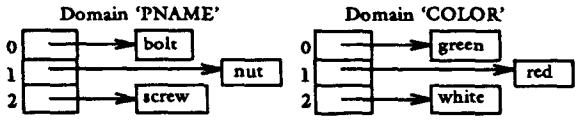
Within a database environment, there are essentially two types of domains: static and dynamic. Static domains are generally quite small and are completely defined at compile time. A dynamic domain is a very large domain such that new tuples tend to add a new value to the domain. As an example, a social security number attribute would require a new domain value for each new tuple. For general purpose database applications, dynamic domains must be supported. Attributes over dynamic domains often require fast access (e.g. through an index) because they serve as user keys. A dynamic array structure, the D-array is currently under development to handle dynamic domains.

But for the purposes of an expert system or even for deductive database query/processing, support for a dynamic domain is not as urgent because the database may not be updated by interactive users. Even in general, large, dynamic domains do not occur very frequently. Based on a very large IDMS database application [Clemson89], 97% of the domains were found to be static and have less than 100 elements. Also, the average size for domain elements was found to be about 20 bytes. For this type of domain, a sorted array is used in the USL MMDBS to store the actual domain values (or pointers to domain values for long or variable- length values). Each distinct value is then uniquely identified by its position in the array, called the offset ID or OID. Relations are stored as a set of arrays: one for each attribute with new tuples added to the end of the arrays.

The most significant advantage of using OIDs that preserve the order of domain values is that it supports a linear sort algorithm. By creating an (empty) linked list for each domain value and then scanning the attribute array, each tuple subscript can be added to the appropriate linked list according to the OID. This has a profound effect on the performance of the query processor, particularly for the join operation. The USL MMDBS is indexless, the linear sort algorithm supports a linear join algorithm and thus is orders of magnitude faster than other main memory database systems.

As an example, consider the relation for parts in Figure 4.1. The pname and color attributes use static domains and both the internal and external representation for the relation is shown in the figure. In Figure 4.2, the sort procedure is shown pictorially with a sort on the color attribute.

Another important characteristic of USL is the simple array structure used to store attributes. First, the array structure facilitates the use of special purpose hardware. The attributes commonly used for a selection can be maintained directly in a content-addressable memory [Colomb87]. Second, for the purpose of both selection and sort, the physical contiguity of individual attributes improves performance both in main memory and in a cache memory. In general, representing a *tuple* in contiguous storage is not necessary. The array structure provides simple, direct access to attributes (and thus to tuples); there is no need to link the tuples together. A third benefit of the array approach is that the tuple subscript can be used directly as a permanent (implicit) identifier. The tuple subscript can be used as a surrogate to support foreign keys and thus support pre-computed joins, as in [Lehman86a]. Also, by adding tuples at the end



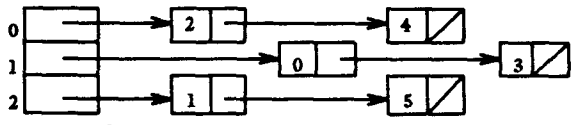
External Representation

pname	color	price
nut	red	10
bolt	white	10
bolt	green	15
screw	red	10
screw	green	15
screw	white	20

Internal Representation

	pname	color	price
0	1	1	10
1	0	2	10
2	0	0	15
3	2	1	10
4	2	0	15
5	2	2	20

The Part Relation
Figure 4.1



Sort Structure for Sort on COLOR
Figure 4.2

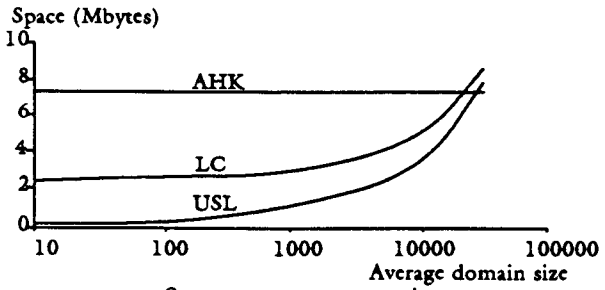
of the array, the tuple subscript serves as an implicit "recency" indicator. This can be used in conflict resolution strategies for production languages such as OPS5 [Brownston85]. Finally, since intermediate results use the same domains, USL creates intermediate relations with the same simple structure. Thus the query processing algorithm (with its strong performance characteristics) can be applied to intermediate relations as easily as base relations.

4.3. Performance of the USL MMDBS

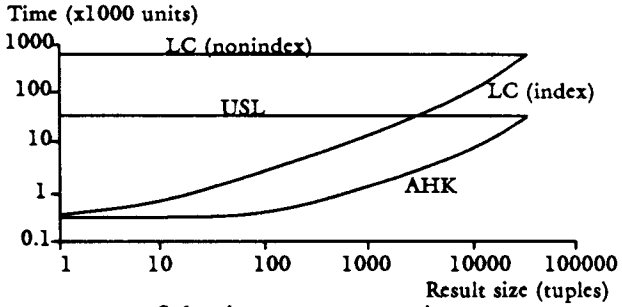
In order to illustrate the performance of our design, we compare the performance of the USL MMDBS with other proposed MMDBS [Ammann85, Lehman86a]. In the remainder of the paper we will refer to these two MMDBSs as AHK and LC, respectively.

Storage Cost: Figure 4.3 shows the storage cost of a relation of 30000 tuples with 10 attributes, where we assume that all domains are static and unique, and the average element size is 20 bytes. We observe that USL is more space efficient than both AHK and LC, especially when the average domain size is small. From Figure 4.3, the space required for USL is only 1/9 and 1/24 of AHK and LC respectively when average domain size is 100 or smaller.

Selection Cost: Figure 4.4 shows the relationship between the selection cost and the size of the result relation. Since USL is indexless, the benefit of the index in AHK and LC is clearly shown when the size of the result is small. However, USL is much better than LC and AHK when the selected attribute is not indexed or selection is performed on a temporary relation. In order to illustrate the effect of the USL selection, Figure 4.5 shows the actual execution times of the USL selection based on our prototype



Storage cost comparison
Figure 4.3



Selection cost comparison
Figure 4.4

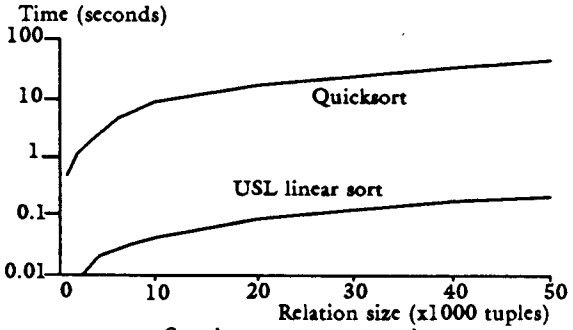
implementation for various relation size. Note that this response time can be achieved for any selection predicate on any attribute on both permanent and temporary relations.

Sorting Cost: Sorted relations can make use of efficient algorithms for set and join operations and for removing duplicate tuples. Figure 4.6 compares the sorting cost between USL linear sort and the quicksort by assuming that the sorted attribute is defined on a domain of size 1000 and the cost of a string compare is 10 times the cost of an integer compare. The linear sort performance of USL is several orders of magnitude better than the quicksort on AHK and LC.

Update Cost: Figure 4.7 summarizes the update costs for all proposed MMDBS. Since USL is

Relational Size (tuples)	Selection Time (seconds)
1000	0.002
10000	0.02
100000	0.2

Experimental results of the USL select operation
Figure 4.5



Sorting cost comparison
Figure 4.6

Database	Update Cost
USL	O(1)
AHK	O(N)
LC (index)	O(logN) + O(M*)
LC (non-index)	O(1)

* M is the node size of the T-Tree.

Update cost comparison
Figure 4.7

indexless, and since, for the moment, all domains are assumed to be static; update is simply replacing the old value with the new value.

5. Analysis and Conclusions

In order to support RPL within the relational DBMS framework, it is necessary to implement the interpreter in main memory. In this paper, the determination of the conflict set, i.e. the match process, is supported through the use of Pattern Match Reduction coupled with the USL MMDBS to support memory-resident query processing. These two components of the RPL research are reviewed here through a comparison with related work. Finally, work in progress is briefly presented.

The use of materialized views to simulate the Rete approach has been suggested by others, notably Hanson [Hanson87, 88] and Sellis [Sellis88]. The distinctive features of the RPL/PMR research are: (1) an RPL rule is strictly more expressive than an OPS5 rule [Delcambre88a], (2) the PMR approach is a generalization of view materialization because the language is relationally complete, because any arbitrary, intermediate node in the query tree can be materialized, and because massive changes (both adds and deletes) are supported against any or all base relations during one interpreter cycle, and (3) the performance comparison reported here is done at an abstract level, without regard for implementation details. The PMR simulation studies serve to clarify the relationship between RPL and the Rete approach.

For a main memory database, the domain-based approach was first proposed for System R [Chamberlin81] but was obviously unsuitable for a disk-resident system. For main memory, the array storage for explicit, static domains provides an excellent compromise for the time/space tradeoffs while supporting a linear join algorithm. For large, dynamic domains the challenge is to explicitly represent the active domain [Maier83] while providing an efficient means for adding new domain values and for searching large domains. We are currently developing a D-array data structure as a compromise between an array and a tree to support such domains.

Other work in progress concerns the further integration of the two research components presented in this paper. The PMR formulas can now be refined in the context of the USL MMDBS data structures and query algorithms. Based on the relative efficiency of the USL algorithms for the relational operations, the PMR formulas may be transformed into an equivalent form that provides better performance. We have the opportunity to consider other query (rule) optimization techniques and also PMR optimizations concerning the choice of nodes for materialization. Other work in progress concerns the migration of data between USL and an conventional relational database in the presence of RPL rule processing.

Finally, Based on the unification of RPL and RDL/1 into a single language that can serve for the purpose of inferencing, deductive database queries, and triggers [Delcambre89], the proposed two level system (using USL and a conventional database) will support a complete expert database system.

References

[Ammann85] Ammann, A., Hanrahan, M., and Krishnamurthy, R., "Design of a Memory Resident DBMS," *Proceedings of IEEE COMPCON Conference*, 1985.

[Bancilhon86] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations," *On Knowledge Base Management Systems*, Springer-Verlag, New York, 1986.

[Blakeley86] Blakeley, J., Larson, P., and Tompa, W., "Efficiently Updating Materialized Views," *Proceedings of ACM SIGMOD Conference*, Washington, D.C., May 1986.

[Brownston85] Brownston, L., et al., *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, Massachusetts, 1985.

[Chamberlin81] Chamberlin, D.D., et al., "A History and Evaluation of System R," *Communications of ACM*, Vol. 24, No. 10, October 1981.

[Clemson89] *IDMS Schema*, from the Information Systems Development Group, Clemson University, Clemson, SC, January 1989.

[Colomb87] Colomb, R.M., "Relational

Operations Using a Bit-Serial Word Parallel Content-Addressable Memory," *TR# TR-FB-87-05*, CSIRO Division of Information Technology, Sydney, Australia, June 1987.

[Delcambre88a] Delcambre, L.M.L. and Etheredge, J.N., "The Relational Production Language: A Production Language for Relational Databases," *Proceedings of the 2nd International Conference on Expert Database Systems*, April 1988.

[Delcambre88b] Delcambre, L.M.L. and Etheredge, J.N., "A Self-Controlling Interpreter for the Relational Production Language," *Proceedings of ACM SIGMOD Conference*, June 1988.

[Delcambre88c] Delcambre, L.M.L., "RPL: An Expert System Language with Query Power," *IEEE Expert*, Vol. 3, No. 4, Winter 1988.

[Delcambre89] Delcambre, L.M.L. and Simon, E., "The Relation Rule Language: A Rule Language for all Database Purposes," in preparation.

[Etheredge89] Etheredge, J., "Pattern Match Reduction in the Relational Production Language," Ph.D. Dissertation, University of Southwestern Louisiana, Lafayette, LA, April 1989.

[Forgy81] Forgy, C.L., "OPS5 User's Manual," Technical Report No. CMU-CS-81-135, Carnegie-Mellon University, 1981.

[Forgy82] Forgy, C.L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, Vol. 19, No. 1, September 1982.

[Hanson87] Hanson, E., "Efficient Support for Rules and Derived Objects in Relational Database Systems," Ph.D. Dissertation, University of California, Berkeley, August 1987.

[Hanson88] Hanson, E., "Processing Queries Against Database Procedures: A Performance Analysis," *Proceedings of ACM SIGMOD Conference*, June 1988.

[Ioannidis88] Ioannidis, Y.E., et al., "BERMUDA - An Architectural Perspective on Interfacting Prolog to a Database Machine," *Proceedings of the 2nd International Conference on Expert Database Systems*, April 1988.

[Lehman86a] Lehman, T.J. and Carey, M.J., "Query Processing in Main Memory Database Management Systems," *Proceedings of ACM SIGMOD Conference*, 1986.

[Maier83] Maier, D., *The Theory of Relational Databases*, Computer Science Press, Rockville, MD, 1983.

[Maindreville88a] de Maindreville, C. and Simon, E., "A Production Rule-Based Approach to Deductive Databases," *Proceedings of IEEE International Conference on Data Engineering*, February 1988.

[Maindreville88b] de Maindreville, C. and Simon, E., "Modeling Non Deterministic Queries and Updates in Deductive Databases," *Proceedings of International Conference on VLDB*, 1988.

[Napheys88] Napheys, B. and Herkimer, D., "A Look at Loosely-Coupled Prolog/Database Systems," *Proceedings of the 2nd International Conference on Expert Database Systems*, April 1988.

[Sellis88] Sellis, T., Lin, C.-C., and Raschid, L., "Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms," *Proceedings of ACM SIGMOD Conference*, June 1988.

[Simon88] Simon, E., and de Maindreville, C., "Deciding Whether a Production Rule is Relational Computable," *Proceedings of the 2nd International Conference on Database Theory*, Bruges, Belgium, September 1988.

[Stonebraker87] Stonebraker, M., Anton, A., and Hanson, E., "Extending a Database System with Procedures," *ACM Transactions on Database Systems*, Vol. 2, No. 3, September 1987.

[Ullman88] Ullman, J.D., *Principles of Database and Knowledge-base Systems*, Computer Science Press, Rockville, MD, 1988.

[Waramahaputi89] Waramahaputi, J. and Delcambre, L.M.L. "USL: The Domain-based Database System for Main Memory Query Processing," in preparation.