The Design of a Virtual Fact Base for Prolog

by

J. Steven Haugh

Project submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

APPROVED:

[Signature]
John Roach, Chairman

[Signature]
Edward Fox

[Signature]
Clifford Shaffer

April 8, 1991
Blacksburg, Virginia
0.2

LD
SLES
VESI
1991
H294
0.2
The Design of a Virtual Fact Base for Prolog

by

J. Steven Haugh

Committee Chairman: John Roach

Computer Science

(ABSTRACT)

The fact and rule list internal to Prolog is capable of handling as many facts as available memory resources permit. A solution to this limitation is to store facts on disk, retrieving them into a main memory database buffer only as needed. Allocating a fixed portion of main memory to buffer database facts frees up scarce main memory for more frequently accessed rules and data structures internal to Prolog. The Prolog Database System built in connection with this project transparently stores and retrieves facts on disk and evaluates them in the order they were asserted allowing for the transfer of existing small scale prototypes into large scale production systems.

Since existing relational database techniques were not designed to function in a Prolog environment where facts are evaluated in a specific order and one at a time, custom
database facilities were designed, developed, and integrated into Prolog. These database facilities include a unique page replacement policy designed to minimize expensive page faults during the execution of a Prolog program. The look ahead page replacement policy looks ahead on database pages while they are in main memory in order to determine whether they are likely to be accessed again in the future. In this way, a near optimal working set of database pages is maintained in the database buffer, assisting with minimizing expensive page faults.
Acknowledgements

I would like to thank Dr. Roach for serving on my committee. I would also like to extend my gratitude to Ph.d candidate, John Deighan, for advice and criticism throughout the course of my research and writing.
Table of Contents

Chapter 1.0 Introduction ........................................ 1
  1.1 Problem Statement .......................................... 1
  1.2 Solution Approach ......................................... 2
  1.3 Project Overview .......................................... 5

Chapter 2.0 Background .......................................... 6
  2.1 Coupling versus Integration ............................... 6
  2.2 Summary .................................................. 11

Chapter 3.0 Analysis .............................................. 13
  3.1 Solution Analysis ........................................ 14

Chapter 4.0 Logical File Structure ............................. 21
  4.1 Logical Data Structures ................................... 23
  4.2 The Relation Table ........................................ 26
  4.3 Asserting a Database Fact ............................... 30
  4.4 The Representation of a Database Fact ............... 31
  4.5 Summary .................................................. 42

Chapter 5.0 Sequential Unification ............................ 44
  5.1 Complexity Analysis ...................................... 48
Chapter 6.0  The Look Ahead Page Replacement Policy ..... 51
  6.1  Look Ahead Unification .............................. 54
  6.2  The Abnormal Termination of a Database Query .. 60
  6.3  The Page Replacement Policy ....................... 63
  6.4  Look Ahead Unification and Indexing .............. 70

Chapter 7.0  Experimental Design and Setup .................. 72
  7.1  The Break Even Analysis Model ..................... 72
  7.2  The Sample Fact Database ........................... 74
  7.3  The Sample Prolog Queries ........................... 80

Chapter 8.0  Empirical Results ............................... 84
  8.1  General Results ..................................... 85
  8.2  Margin of Improvement Analysis ....................... 92
  8.3  Results Summary .................................... 98

Chapter 9.0  Summary and Concluding Remarks ................. 105
  9.1  Partial Look Ahead Matching .......................... 105
  9.2  Project Summary .................................... 114

Bibliography .................................................. 116
List of Illustrations

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Control Flow</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Page Table and Database Buffer</td>
<td>19</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Relation Stack</td>
<td>22</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Relation Table</td>
<td>27</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Database Fact</td>
<td>32</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Unification Algorithm</td>
<td>34</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Assert Algorithm</td>
<td>43</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Initial Goal Unification Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Goal Unification Algorithm</td>
<td>47</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Look Ahead Algorithm</td>
<td>58</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Database Generation Program</td>
<td>77</td>
</tr>
<tr>
<td>Table 1</td>
<td>Query Statistics</td>
<td>82</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Representative Page Fault Range</td>
<td>83</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Execution Time Relative to Page Size</td>
<td>86</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Execution Time Relative to Buffer Size</td>
<td>87</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Page Faults Relative to Page Size</td>
<td>88</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Page Faults Relative to Buffer Size</td>
<td>89</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Average Page Fault Time</td>
<td>91</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Time Savings</td>
<td>93</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Page Fault Savings</td>
<td>94</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Look Ahead Success Relative to Page Size</td>
<td>95</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Look Ahead Success Relative to Buffer Size</td>
<td>97</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Look Ahead Effectiveness Ratio</td>
<td>99</td>
</tr>
</tbody>
</table>
Figure 23  Results Relative to Page Size ..................... 100
Figure 24  Results Relative to Buffer Size ................... 101
Figure 25  Overall Results ...................................... 102
Figure 26  Page Fault Savings Averages ....................... 109
Figure 27  Time Savings Averages ............................... 110
Figure 28  Page Fault Savings Variances ...................... 111
Figure 29  Time Savings Variances .............................. 112
Figure 30  Preliminary Results ................................. 113
Chapter 1.0 Introduction

Prolog facts containing identical predicates and the
tuples of a relation in a relational database have similar
properties. In a relational database, a tuple of a
particular relation contains a set of attribute values much
like a Prolog fact contains a set of argument values.
Storage techniques used in existing relational database
systems can be modified and incorporated into the design of
a database facility for Prolog. The system resulting from
our efforts in this direction is referred to as the Prolog
Database System.

1.1 Problem Statement

Not all aspects of a relational database system are
adaptable to Prolog. The major difference between Prolog
fact evaluation and relational tuple evaluation is that
Prolog evaluates facts in a specific order and one at a time
while a relational database evaluates unordered tuples a set
at a time. Set retrieval techniques incorporated into
relational databases do not offer the granularity of
evaluation that Prolog requires.

Page replacement policies found in relational database
systems were not designed to function in a Prolog
environment. They do not take advantage of all information available in the Prolog environment to avoid page faults. Once Prolog unifies a goal with a database fact, there is information available on that same page that can be used to avoid future page faults. This information is the rest of the facts on that page. For instance, if the next unifiable fact for the goal is on the page, then that page usually will be accessed again in the future if Prolog backtracks to the current goal.

1.2 Solution Approach

This project is based on the hypothesis that existing database management techniques can be modified and integrated into Prolog to extend the fact storage and retrieval capabilities of Prolog efficiently. A database facility was designed and integrated into Prolog to form the Prolog Database System. The design incorporates existing relational database storage techniques [8][10][24][25] such as virtual relation stacks that hold all of the database facts containing the same predicate. The design also incorporates non-standard database techniques designed to meet the special needs of Prolog such as:
1) One fact at a time evaluation
2) Ordered fact evaluation
3) Specialized page replacement policy

To meet the "one fact at a time evaluation" criterion, the database facility makes use of the Prolog stack [6][7][29]. Once a database fact has been unified with a goal, information is stored on the Prolog stack so that the current state of the database facility can later be restored. Prolog also uses the stack to represent the current state of the Prolog Machine. As Prolog executes, new variable bindings are represented by pushing environments onto the stack. When Prolog backtracks, these environments are popped from the stack restoring a previous state of the Prolog Machine. The database facility uses the stack for the same purpose. When Prolog backtracks to a goal, the database facility is restored to a previous state so that goal unification can resume from where it was previously suspended.

To meet the "ordered fact evaluation" criteria of Prolog, a stack oriented fact storage scheme was implemented. A stack maintains the temporal order of fact assertion so that the facts can be evaluated in that order. This approach supports both ordered and order independent
relations.  

Many Prolog programs rely on the order of fact evaluation to determine the order in which answers are returned. For instance, database facts describing an attribute of an object may be asserted in a temporal sequence. To access only the most recent value of that attribute, a cut goal may be used. These types of Prolog programs may be ported to the large scale Prolog Database System without modification as fact storage needs grow.

To meet the "optimal page replacement policy" criterion, the database facility maintains data structures necessary to describe the likelihood of future access to the database buffer pages. When it is determined that a database page will be accessed again in the future, the status of that page is updated to signal the page replacement policy that the page should be retained in main memory. A potential page fault is avoided since this page may have otherwise been paged out to disk before it was accessed again.

The integration of database facilities into Prolog is achieved by:

1) Using the Prolog stack to save the state of the database facility.
2) Maintaining the temporal order of fact assertion on stacks.

3) Looking ahead on pages while they are in main memory to determine whether they are likely to be accessed again.

1.3 Project Overview

Background information is presented in Chapter 2 and Chapter 3 provides an analysis of the problem with a terminology overview. Before the page replacement policy is discussed, the logical structure of the database file is presented in Chapter 4. An analysis of the integration of the database facility into Prolog is presented in Chapter 5. With a basic understanding of the relationship between the Prolog Machine and the database facilities, the look ahead page replacement policy is presented in Chapter 6. Chapters 7 and 8 present tests and results that are intended to determine the efficiency and effectiveness of the look ahead page replacement policy for a sample database and query set. Finally, Chapter 9 summarizes the project and future research directions.
Chapter 2.0 Background

Previous approaches toward extending the fact storage and retrieval facilities of Prolog onto disk have included homogeneous and heterogeneous approaches [14]. The homogeneous approach involves the integration of database facilities into Prolog resulting in a single process system. The heterogeneous approach involves the coupling of Prolog with an existing DBMS resulting in two separate processes connected via communication channels such as UNIX pipes.

2.1 Coupling versus Integration

Coupling Prolog with an existing DBMS is the more common approach since the integration alternative involves the development of database facilities. Another advantage of coupling is that the existing DBMS that is coupled with Prolog can continue to provide services such as data security, data integrity, and concurrency control. The Prolog process and the DBMS process may execute on the same or different host computers linked by means of communication channels such as UNIX sockets, pipes, or files.
2.1.1 Coupling

Two approaches can be taken toward coupling Prolog with an existing DBMS, the close approach and the loose approach [2]. Close coupling preserves the Prolog interface to the user while loose coupling departs from the traditional Prolog interface.

The purpose of close coupling is to hide transparently the underlying DBMS from the user. When a Prolog goal requiring database access is encountered, a process and a set of pipes are set up to connect Prolog with the DBMS. The new process generates an equivalent database query in the data manipulation language of the underlying DBMS which is sent down the request pipe to the DBMS. Once the database query has been executed by the DBMS, the resulting relation is piped back to the appropriate Prolog goal process through a reply pipe. The reply pipe acts as a queue of answers retrieved one at a time as Prolog backtracks. The extensive use of processes and pipes obviously imposes a heavy burden on the operating system. This method does offer a way to handle recursion, although the depth of recursion is limited by the number of pipes available from the operating system.

The coupled portion of the Nu-Prolog [19] Deductive Database System use a similar strategy to access data in an
external DBMS. Nu-Prolog maintains the state of a database query by saving backtracking information on the Prolog Stack. This information includes a file identifier referencing the disk file containing the answers to the query as generated by the external DBMS. One disadvantage is that the facts are represented within the DBMS in textual format. A large amount of time is spent parsing the facts into the internal representation expected by Prolog. If, however, the database facts were stored in a Prolog format within the DBMS, then other applications would have to translate them into textual format.

Query optimization can reduce the number of database queries generated by delaying the actual execution of Prolog goals as long as possible until a more efficient relational database query may be formulated [27]. The queries for multiple Prolog goals are consolidated into one relational database query. If the set of answers is not too large, then they may be asserted into the internal Prolog database for evaluation.

The purpose of loose coupling [2] is to avoid the overhead associated with query optimization by supplementing Prolog with additional features. The user is provided with non-Prolog methods of expression that exploit the efficient set retrieval techniques of the underlying DBMS making the interface between Prolog and the DBMS more efficient.
As an example, consider the following Prolog query requiring the retrieval of all employee facts:

( employee ?name ?salary )
( > salary 5000 )
( fail )

Although the one fact at a time retrieval method of Prolog is logically effective, it is also inefficient in cases such as this. By extending the Prolog syntax, this query may be condensed to exploit the retrieval techniques of the underlying DBMS so that all employee facts are not retrieved [9]:

( employee ?name ?salary > 50000 )

To extend this idea further, the user can be allowed to embed a database query for the underlying DBMS directly into Prolog code:

SQL: select employee where salary > 50000

The major disadvantage of this approach is that data manipulation languages do not allow a means for expressing recursion. A Prolog clause that refers to itself cannot be represented in standard data manipulation languages.
2.1.2 Integration

Although close coupling without query optimization allows the user to express recursion, the depth of the recursion is limited by the number of pipes and processes that the operating system can supply. Integration, on the other hand, allows the user to express recursion without those inter-process communication problems associated with coupling.

The EDUCE system combines loose coupling for non-recursive queries with integration for recursive queries [2]. EDUCE accesses the database either through the integrated method or the loosely coupled method depending on the recursive nature of the query that is to be satisfied. EDUCE evaluates recursive Prolog clauses using the integration approach where the record level access method of the underlying DBMS is integrated into Prolog. This integrated portion of EDUCE is free from the inter-process communication problems associated with coupling since the database system is integrated directly into Prolog. To retain the efficient set retrieval techniques of the DBMS, EDUCE uses loose coupling to satisfy non-recursive queries.

The Nu-Prolog Deductive Database System has both a coupling to an external DBMS and an internally integrated database [19]. In the integrated database, partial match
queries are satisfied using a two level superimposed codeword hashing scheme [19]. The top level locates all database segments (pages) that may contain matching facts while the second level locates the possibly matching facts on those pages. Queries containing conjunctions of Prolog goals are optimized by taking advantage of the way the facts are stored. Database facts are not stored in the order they were asserted, so they can be organized in a way that minimizes page faults. The "Superjoin Algorithm" [19] attempts to satisfy a conjunction of goals by partitioning the database into disjoint regions so that each page is accessed only once if buffer space permits. This is achieved by storing facts on pages based on the indexing scheme used and by using the internal fact and rule list of Prolog as a buffer. Recursive queries are optimized using a "return all answers" [19] method that generates a set of tuples that includes all possible answers.

2.2 Summary

The coupling approach is attractive due to the reuse of existing database facilities. Although functional needs are affected with little development effort, operating system resources are strained limiting efficiency. The integration approach overcomes heavy dependence on operating system
resources but requires more development effort. However, neither approach addresses the need for Prolog to evaluate facts in the order they were asserted.

The purpose of the research presented in this project paper is to design, develop, and integrate database facilities into Prolog that are completely transparent to the Prolog programmer. This transparency includes the evaluation of database facts in the order they were asserted. The internal database integrated into Nu-Prolog operates on an unordered set of facts allowing for many beneficial optimization techniques. This project paper will present approaches to database integration that promote a temporal relationship among the facts based on their order of assertion.
Chapter 3.0 Analysis

When a Prolog knowledge base contains more facts than can be stored in main memory, the facts must be stored on disk and paged into the main memory database buffer as they are requested. Main memory is generally just physical random access memory (RAM) unless the firmware and operating system implement virtual memory. Virtual memory is a logical extension of physical memory where some of the main memory pages are actually stored on disk and paged into physical memory as they are requested. Even though virtual memory extends the address space of main memory, virtual memory page faults degrade the performance of the Prolog Database System.

Given this, database facilities must be integrated into Prolog that provide access to facts stored on disk in the order they were asserted. These facilities must provide a paging service that maintains a working set of the database pages in a special main memory database buffer. An efficient and effective page replacement policy is required that takes advantage of all available information to minimize expensive page faults. These facilities must also include an indexing service that reduces the number of facts retrieved from disk for unification purposes.
3.1 Solution Analysis

Figure 1 shows the architecture of the Prolog Database System. I designed and implemented the Database and Page Managers. I also specified interface requirements for the Index Manager. Ph.d candidate John Deighan at Virginia Tech, developed the Prolog Machine and the Index Manager. Once the original version of the Prolog Database System was complete, I retained a copy for page replacement policy research. Since that time, Mr. Deighan has made modifications to his version for production use. In the remainder of this project paper, I will be referring to my original version unless otherwise specified.

3.1.1 Database Manager

When Prolog encounters a goal containing a database predicate, the goal is passed to the Database Manager where an attempt is made to unify the goal with a database fact. This event is referred to as the first instantiation of the database query. The next instantiation of the database query occurs when Prolog backtracks to the goal. The process of unifying a goal with all unifiable database facts is referred to in this project as processing a database query.
Figure 1

Control Flow
The lifetime of a database query is defined as the number of times a database query is instantiated. The lifetime of a database query is directly related to the number of database facts that will unify with the goal which is directly related to the number of unbound variable arguments in the Prolog goal. More unbound variable arguments imply a longer database query lifetime since the database query is less constrained.

3.1.2 Index Manager

The purpose of the Index Manager is to reduce the number of database facts that must be retrieved from the database for attempted unification with the Prolog goal. The ultimate indexing scheme would produce a list of addresses representing all of the database facts that will unify with a Prolog goal in the order those facts were asserted. The indexing schemes considered for the Prolog Database System, however, only index on one argument of the Prolog goal. These indexing schemes include perfect hashing, B+ trees, and superimposed coding [18].

The indexing technique used in this version of the Prolog Database System indexes on one argument of the Prolog goal and produces a list of possibly unifiable database facts in the order they were asserted. The perfect hashing
technique [18] is used to generate a reduced search space consisting of a list of all database facts containing an argument that will unify with the corresponding argument of the Prolog goal and a database predicate matching the database predicate of the Prolog goal. The resulting search space is used to reduce the number of database facts with which unification is attempted.

The number of fact addresses generated by this indexing technique is related to the distribution of atoms throughout the database. Indexing on an argument bound to a frequently occurring atom will generate more database fact addresses than indexing on an argument bound to a less frequently occurring atom. For an indexing scheme that can index on one argument position, such a perfect hashing [18], the expected number of fact addresses generated in the reduced search space can be estimated as the average number of times that any particular atom is expected to occur in that argument position. Once index access is complete, the Database Manager requests pages containing those database facts identified in the reduced search space in the order they were asserted.

3.1.3 Page Manager

The Page Manager is responsible for the storage and
retrieval of database facts. The database facts are stored on fixed size pages in the disk file and are copied into the main memory database buffer when they are requested. Each frame of the database buffer holds one database page. The number of frames in the database buffer can be adjusted, making the Prolog Database System portable to target machines with differing amounts of main memory.

The Page Manager also maintains a page table. The page table is used to determine which database page is in a particular database buffer frame and whether the database page in the frame has been modified (i.e., "dirty" bit set) since it was read into the frame (Figure 2).

The Page Manager is responsible for retrieving pages into the database buffer as they are requested. When a page is requested, the Page Manager searches the page table to determine whether the requested page is already in the database buffer. The page table is currently searched sequentially, however, faster search capabilities are planned. If the requested page is in the database buffer, then the identifier for the database buffer frame holding the page is returned to the requestor. If the requested page is not found in the page table, then a page fault is generated to read the requested page from disk into a frame of the database buffer.
Figure 2
Page Table and Database Buffer
When a page fault occurs, space must be made available within the database buffer to hold the requested page. To make space, a database page in the database buffer must be selected for replacement. The rules used to select the page for replacement are collectively called the page replacement policy [24].

If the page selected for replacement has not been modified since it was read into the database buffer, then the requested page can be copied into the replacement frame overwriting the replacement page [10]. If the replacement page has been modified since it was read into the database buffer, then it must be written to disk before the requested page can be read into the replacement frame [10]. This is an expensive page fault since it involves two page transfers.

The I/O activity associated with a page fault is expensive in terms of disk access time and cpu instruction time. It is therefore important to minimize page faults [10] with a well designed page replacement policy and large database pages with database facts packed densely.
Chapter 4.0 Logical File Structure

A fact storage scheme designed to meet the specific needs of Prolog was incorporated into the Prolog Database System. A relational approach was taken toward the structure of the fact database where the database file is partitioned into relations that store all of the database facts with a particular database predicate. The fact database is stronger than relational since the facts within a relation are chained together in the order they were asserted. The set of tuples in a relation of a relational database have no ordering.

The Page Manager stores all of the database facts with the same database predicate on a relation stack. The database facts are pushed onto the stack as they are asserted maintaining their temporal relationship. One relation stack exists for every database predicate and is implemented by chaining database pages together (Figure 3).

In the absence of indexing, a database query is satisfied by attempting to unify the goal with every database fact on a relation stack. The corresponding relation stack is traversed from bottom to top, one database page at a time. Only a portion of the relation stack is resident in the main memory database buffer at any given time.
Figure 3
Relation Stack
Section 4.1 defines what a logical data structure is and how it is maintained. Section 4.2 describes a logical data structure called the relation table which is used to locate a particular relation stack. Section 4.3 and 4.4 describe the assertion of a database fact and the internal representation of a database fact on a database page. The storage scheme allows random access to any argument of a database fact without the necessity of traversing the preceding arguments of that fact. This approach allows for the design of efficient unification strategies.

4.1 Logical Data Structures

A logical data structure is simply a chain of database pages used to store information. A logical data structure can be used to virtually implement data structures such as stacks and tables. The fact database is a disk file physically divided into fixed size database pages each of which fits into a database buffer frame. Each database page has a pointer field identifying the page number of the next page in the chain. A page number (Figure 3) is an offset of a page relative to the beginning of the database file beginning with page number one.

The maximum length of a logical data structure is constrained by the maximum file size allowed by the
operating system or the amount of disk space available. Logical data structures of any size can be accessed one page at a time by means of a page fault into the main memory database buffer.

The logical file structure of the facts database file consists of relation stacks, the free page stack, and the relation table. The Page Manager has exclusive control over these logical data structures.

4.1.1 Logical Data Structure Maintenance

Database pages must be available for allocation as logical data structures expand. For example, a new database page must be allocated when a database fact is asserted onto a relation stack and the top page of that page chain is full. Database pages must also be released as logical data structures contract. The page numbers of released and allocated pages are pushed onto and popped from the free page stack. Reclaiming released pages reduces wasted space within the database file.

The built-in retract predicate is used to delete the last fact from a relation. A database page is deallocated when all of the data on the top page of a relation stack has been deleted. Eventually, a general retract predicate will be implemented requiring a compaction capability in order to
reclaim unused space within a relation stack.

The number of the most recently deallocated page is on top of the free page stack and is returned as the reply to the next page allocation request. A new database page is allocated from the end of the database file when a page is requested and the free page stack is empty. Database pages may be allocated from the end of the database file as long as there is space on the disk and the operating system can provide them.

From the free page stack algorithm, it is clear that the pages in a page chain will not necessarily be stored logically adjacent within the database file. The location of the next page in a page chain depends upon the location of the page on top of the free page stack. Even if the pages making up a page chain are logically adjacent within the database file, they may be physically located at different locations on the disk depending on the file system of the host operating system. The Unix operating system stores portions of the database files at locations on the disk where space is available.

A proposed solution to this problem is to maintain each relation independently. The raw disk interface can be used to bypass the Unix file system [25] allowing the user to allocate partitions of the disk to the relations of the database. The raw disk interface allows the user to access
the partitions of the disk directly so that successive pages of the relation can be written to the disk contiguously, thus reducing disk arm movement when the pages of the relation are accessed sequentially.

4.2 The Relation Table

The relation table is a logical data structure that serves to record the locations of the relation stacks in the fact database. For every database predicate, there exists a unique entry in the relation table linking it to the corresponding relation stack where all of the facts containing that predicate are stored (Figure 4).

At this point in time, all database facts with a particular database predicate must have the same number of arguments. Eventually, the system will be extended to identify a relation by means of the database predicate and the number of arguments treating each combination as a different predicate.

Top and bottom of stack pointers are provided in the form of a page number and an offset within that page. The offset is necessary since the top of stack may fall in the middle of the top page of the relation stack. An explanation of the link field is forth coming.
<table>
<thead>
<tr>
<th>Entry</th>
<th>Relation Name</th>
<th>Bottom Page</th>
<th>Bottom Offset</th>
<th>Top Page</th>
<th>Top Offset</th>
<th>Link Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Utilizes</td>
<td>354</td>
<td>000</td>
<td>100</td>
<td>876</td>
<td>650</td>
</tr>
<tr>
<td>#2</td>
<td>Isprocess</td>
<td>876</td>
<td>000</td>
<td>752</td>
<td>4877</td>
<td>999</td>
</tr>
<tr>
<td>#3</td>
<td>Iselement</td>
<td>667</td>
<td>000</td>
<td>333</td>
<td>6623</td>
<td>777</td>
</tr>
<tr>
<td>#4</td>
<td>Employs</td>
<td>87</td>
<td>000</td>
<td>356</td>
<td>889</td>
<td>337</td>
</tr>
</tbody>
</table>

Page 748

Figure 4
Relation Table
The database facts of a relation stack are accessed sequentially by traversing the page chain of that relation stack from bottom to top. The bottom of stack pointer in the relation table entry points to the first database fact pushed onto the relation stack. The top and bottom of stack pointers both point to the base of the bottom page of the relation stack when there are no database facts on the stack. The bottom of stack pointer is updated in the event of an assert_first built-in predicate just as the top of stack pointer is updated in the event of an assert_last built-in predicate. The default for assert is assert_last.

4.2.1 Declaring a Database Predicate

An entry is created in the relation table when a database predicate is declared. The locations of the top and bottom of a newly created relation stack are stored in the relation table entry. All database facts containing the new database predicate will be pushed onto this stack as they are asserted.

This portion of the Prolog Database System was designed and implemented by the Virginia Tech researcher in charge of the Prolog Machine since it is the Prolog Machine that is responsible for the identification of database predicates during the execution of a Prolog program. The Prolog
Machine maintains a list of special database predicate atoms that trigger the generation of database queries.

One alternative considered was to allow the Prolog Machine to dynamically declare database predicates when the number of facts in the internal fact and rule list exceeds a threshold value. The problem with this approach is that the Prolog system would have to suspend execution when the threshold is exceeded while the database relation is set up and the facts transferred. Even though the disruption could have been minimized by launching the transformation as a background process, this implementation was postponed to a later date as an enhancement. Instead, it was decided that the user must know ahead of time what sets of facts should be stored in the database. Therefore, declaration of database queries is the responsibility of the user. Two methods are available for the declaration of a database predicate.

One way to declare a database predicate is to assert a set of facts where the first fact has a predicate with a ":database" suffix:

```
( assert ( ( clothes: database blue shoe ) )
  ( ( clothes red hat ) )
  ( ( clothes orange glove ) ) )
```

A user may also use a special Prolog command, "loaddb
"<filename>"", to declare a database predicate automatically and assert many facts onto a relation stack (Ex. "loaddb clothes"). The Prolog machine looks for the disk file called <filename>. If <filename> exists, then it is declared as a database predicate and all database facts in that file are asserted onto the relation stack for that database predicate. If <filename> does not exist then the user is notified of the error. The arguments of each database fact in <filename> must be stored on a line by themselves with a space between each argument. This method is intended to serve as an external interface to an existing DBMS. Once a database predicate has been declared, all facts asserted containing that predicate will be asserted into the fact database.

4.3 Asserting a Database Fact

The Prolog Machine checks the predicate of a fact when it is asserted. If the predicate has been previously declared as a database predicate, then the fact is sent to the Page Manager for assertion. The relation table is searched for the entry corresponding to the database predicate so that the top of the appropriate relation stack can be located. The appropriate entry is located using a simple sequential search since the expected number of
database predicates is expected to be relatively small. A more efficient search technique should be implemented before the system goes into production to handle databases with large numbers of database predicates.

The arguments of the fact to be asserted are available to the Page Manager through the argument registers of the Prolog Machine. The arguments of the fact are stored in the database as they are dereferenced. The version of the Prolog Database System used for page replacement policy research only stores arguments of type atom.

4.4 The Representation of a Database Fact

A database fact is represented within a relation stack as a set of argument values stacked on top of a link slot stacked on top of a set of argument slots (Figure 5). An argument value and argument slot exist for each argument of the database fact.

In order to avoid unnecessary page faults, each entire database fact is stored on the same database page. Although this limits the size of a fact that can be stored to the size of a database page, it avoids page faults since two pages could potentially be accessed in order to examine one fact if a database fact were split across two database pages possibly resulting in two page faults.
Figure 5

Database Fact
The Prolog Database System incorporates a scheme for the storage of database facts that allows for efficient unification strategies. Database facts are stored in a way that allows access to any argument of a database fact without traversing any other arguments of that fact. This is accomplished by storing an argument slot for every argument of a fact (Figure 5). The address of the first byte of the first argument slot is the address of that database fact. Each argument slot contains the type of the corresponding argument and a pointer to the location on the page where the value for that argument is stored.

These arguments slots can be randomly accessed since the argument slots are all the same size and the number of argument slots is maintained by the Prolog Machine. Using this scheme, the Database Manager compares the argument types of the fact with the argument types of the goal (Figure 6). If the types do not match, then unification will obviously fail, otherwise, unification is attempted. The order in which the argument values of the database fact are accessed for attempted unification is flexible since they can be accessed randomly.
procedure Unify( goal; fact )
  for each argument slot
    compare argument types
    if not successful then
      return fail
    end if
  end for
  for each argument value
    compare argument values
    if not successful then
      return fail
    end if
  end for
  return succeed
end Unify

Figure 6
Unification Algorithm
When the storage of complex argument types such as lists are incorporated into the Prolog Database System, the Database Manager will have the flexibility to attempt unification of simple atom arguments before attempting unification of complex list arguments. An attempt to unify an atomic argument will generally require fewer comparisons than list arguments, allowing the Database Manager to determine whether a database fact will unify with a goal with fewer comparisons since the arguments of the database fact can be accessed randomly.

4.4.1 Argument Slot

Each argument slot is a four byte quantity partitioned into two fields. The first two bytes identify the type of the database argument thus allowing many possible argument types. The remaining two bytes identify an offset within the page pointing to the first byte of the value for that database fact argument, thus, allowing for a sixty four kilobyte page size. The size of the database pages can be adjusted up to the maximum page size at compile time by adjusting a Database Module page size constant.

Modifications to the storage structure are necessary to allow for a larger database page size. A larger argument slot, however, implies more storage space per database fact.
More storage space per database fact implies that fewer database facts can be stored on a page thus increasing the likelihood of page faults during the satisfaction of a database query.

4.4.1.1 Argument Type

Although the Page Manager is only capable of storing atom arguments at the present time, modifications could eventually allow for the storage of strings, numbers, unbound variables, and lists. The storage, unification, and indexing of lists and unbound variables would be more complex than for atoms.

The storage of unbound variables in the database would be complicated by the possibility of a database fact containing the same unbound variable in different argument positions. One solution would be to store a unique integer code for each unique unbound variable in the asserted database fact. The unique integer code need only be local to that particular database fact. If two different arguments of the database fact contain the same unbound variable, then the value slot for those arguments would contain the same integer code.

The storage of lists in the database would require that the Prolog representation for lists be translated into a
packed representation that could be written to a value slot on a database page. One solution would be to store lists in a sequential format using a car-cdr approach with recursion to handle nested lists. It should be pointed out that a relation stack with lists would not be in first normal form since a relation of a relational database contains only atomic values.

4.4.1.2 Argument Value Offset

When a database fact is to be pushed onto a relation stack, the Page Manager computes the amount of space needed to store that fact and the amount of space available on the top page of the relation stack. The fact will be stored on the top page of the relation stack when there is enough space on that page to store the entire fact. If there is not enough space available on the top page, then a new top page will be allocated to the relation stack so that the entire fact can be stored on a single page.

This implies that there will be wasted space at the end of a database page when there is not enough space available on the page to store the next entire database fact. Only a small portion of the page is wasted however since the size of a database fact is kept small by storing pointers to atom strings instead of the variable length atom strings.
This compact representation also makes it unlikely that a database fact will ever be larger than a database page.

If an argument value (the pointer to the atom string) for a database fact were stored on a different page than the corresponding argument slot, then the argument value offset would have to be in the form of a page number and an offset within that page. By storing an entire fact on a single page, the argument value offset is more compact since it is only an offset within the current page. The potential for wasted space at the end of the database page is offset to a certain extent by the space savings realized with an argument value offset in the form of an offset within the current page as opposed to an argument value offset in the form of a page number and an offset within that page.

4.4.2 Link Slot

The link slot of a database fact is stored immediately after the argument slots for that fact. If a database fact fails to unify with a goal, then the link slot is accessed to locate the next database fact on the page. The link slot is readily accessible since the size and number of argument slots are known. The link slot is the two bytes immediately following the last argument slot of a database fact. If the database fact is the last fact on the page, then the link
slot will contain a zero, indicating that the next page of
the relation stack contains the next database fact to be
accessed.

4.4.3 Argument Value

The argument value for the first argument of the
database fact is stored immediately after the link slot.
The location of the first byte of the first argument value
is calculated by multiplying the argument slot size by the
number of arguments and adding the size of the link slot.
This result is stored as the argument value offset of the
first argument slot.

Once the position of the first argument value is
calculated and stored in the first argument slot, the value
of the first argument is extracted from the argument
register of the Prolog Machine and written onto the page at
the calculated position. In this way an argument slot and
the corresponding argument value may be written to the page
at the same time.

Once the first argument slot and argument value have
been written, the offset yielding the value of the second
argument value may be calculated by adding the size of the
first argument value to the offset calculated for the first
argument slot. In this way, all argument slots and argument
values may be written to the page.

Then the new top of the stack offset is known and can be written to the link slot. The link slot points to the position where the next database fact will be stored, if there is enough space left on the page. Until the next fact is asserted, the Page Manager does not know if the next fact will be stored at the current top of stack or if the next fact will be stored on a newly allocated page where enough space is available to store the entire fact. When the next fact is asserted, the link slot may need to be updated. The address of the link slot of the database fact on top of the relation stack is stored in the relation table so that it may be updated if the necessity arises.

4.4.4 The Internal Representation of Atoms

When a fact is asserted into Prolog, whether it is a database fact or not, all arguments of that fact of type atom will be replaced by the Prolog Machine with a unique integer representing the atom. The Prolog Machine employs database techniques to maintain the mapping of a large number of atoms to integers.

If a new atom is encountered that does not already have a unique integer assigned to represent it, then the atom string will be stored in the Prolog atom table and a unique
integer will be assigned to represent it for the life of the atom. This unique integer will be stored in the fact database in place of the actual atom string. This method of atom storage minimizes the amount of space necessary to store a database fact. No matter how long the atom string is, it is represented by a four byte integer.

More database facts can be packed onto a database page when the amount of space necessary to store a database fact is reduced. Packing more database facts on a database page implies fewer page faults during the satisfaction of a database query. The unique integer representation for atoms also reduces the amount of computation necessary to compare two atoms during the unification process. The actual atom string will only be needed when the atom is to be printed out.

A database predicate is also a special type of atom that is represented as a unique integer. By replacing the database predicate atom strings in the relation table with unique integers, the relation table can be searched more efficiently. By decreasing the storage space needed for a relation table entry, more entries can be stored on a database page of the relation table, thus resulting in fewer potential page faults.

Although the unique integer representation does require more effort when a fact is asserted and when an atom is to
be printed out, the benefits are realized in terms of reduced storage space and reduced comparison time.

4.5 Summary

All logical data structures of the fact database are utilized in the assertion of a database fact. Once the Database Manager receives a goal from the Prolog Machine, it passes the goal on to the Page Manager for the assertion as described in Figure 7.

The file structure of the fact database was designed to meet the specific needs of Prolog as outlined in this chapter. A Prolog Machine coupled with an existing Database Management System is not afforded this luxury since the existing DBMS was designed without regard to the potential interface with Prolog.
procedure Assert( goal )
    search relation table for database predicate entry
    get top of stack pointer
    retrieve top of stack page
    if goal will not fit on the page then
        allocate a new top of stack page
        update the link slot of the top fact
    end if
    for each argument of the goal
        write the argument slot to the page
        write the argument value to the page
    end for
    write the link slot to the page
    update the relation table entry:
        update the top of stack pointer
        update the link slot pointer
    mark the page as dirty
    return to Prolog via the Database Manager
end Assert

Figure 7
Assert Algorithm
Chapter 5.0  Sequential Unification

The Database Module is designed to satisfy database queries with or without indexing services. The need to sequentially satisfy a database query without indexing arises when, for example, the Prolog Machine encounters a goal consisting of a database predicate and arguments, none of which are bound. Under these circumstances, every database fact containing that database predicate must be accessed for attempted unification during the satisfaction of the database query. All of the database facts with a particular database predicate are stored on a relation stack within the database file. The benefits of the previously described database file structure may be exploited to satisfy this type of database query.

The Index Manager implemented in the current version of the Prolog Database System uses perfect hashing [18] and, thus, offers no advantages under these conditions. Perfect hashing is only effective when at least one of the arguments of a goal is bound.

During the first instantiation of this sequential database query, the Database Manager is responsible for the unification of the goal with a database fact (Figure 8). The Page Manager uses the database predicate of the goal as an index into the relation table.
procedure Get_First( goal; backtrack_point )

search relation table for database predicate entry
get the bottom and top of stack pointers
for page = bottom page .. top page of stack
request page from Page Manager
for fact = first fact .. last fact on page
if top of stack then
    return to Prolog for deep fail
else
    attempt unification with fact
    if successful unification then
        find backtrack point
        return unified goal and backtrack point to Prolog
    end if
end if
end if
end for
end for
end Get_First

Figure 8
Initial Goal Unification Algorithm
Once the appropriate entry is located, the bottom of stack pointer is used to request the bottom page of the relation stack from the Page Manager. Using link slots to traverse the database facts on a page and the page pointer links to traverse the pages of the relation stack, the Database Manager attempts to unify the goal with the database facts on the relation stack until successful. If none of the database facts on the relation stack will unify with the goal, then the life of the database query ends and the Database Manager returns control to Prolog where a deep fail occurs, triggering backtracking. However, if the Database Manager successfully unifies the goal with a database fact, then the unified goal is returned to Prolog along with the address of the next fact on the relation stack. This address is referred to as the backtrack point for the database query.

Prolog creates a choice point on the Prolog Stack for the backtrack point so that the state of the database query may be saved. Once this is done, the Prolog Stack will grow as execution of the Prolog program continues. Eventually, Prolog may backtrack to this goal and the Prolog Stack will have contracted back to this choice point. The backtrack point is then returned to the Database Manager along with the goal so that the database query may resume from where it was previously suspended (Figure 9).
procedure Get_Next( goal; backtrack point )
    request page containing backtrack point
    for page = backtrack page .. top page of stack
        for fact = backtrack fact .. last fact on page
            if top of stack then
                return to Prolog for deep fail
            else
                attempt unification with fact
                if successful unification then
                    find backtrack point
                    return unified goal and backtrack point to Prolog
                end if
            end if
        end for
    end for
end Get_Next

Figure 9
Goal Unification Algorithm
This cycle of events continues until the Page Manager reaches the top of the relation stack and there are no more database facts left for unification attempts. At this point, all answers to the database query have been returned and the Prolog Machine executes a deep fail terminating the life of the database query.

5.1 Complexity Analysis

To understand the implications of integrating database facilities into Prolog, we must analyze the complexity of the system. Assuming \( N \) database facts in each relation, the complexity of the satisfaction of a sequential database query is \( O(N) \) since a sequential database query necessitates unification attempts with every fact in a relation stack.

To understand the complexity of the overall system, we must determine the total number of database queries that will be generated for a Prolog query. For simplicity, let us assume that there are \( M \) goals in a Prolog query. The Prolog search space forms a tree where the root node represents the choice point for the first goal. If we assume that every goal in the Prolog query will unify with \( N_U \) of the database facts, then the second level of the tree consists of \( N_U \) choice points for the second goal, the third level of the tree consists of \( N_U^2 \) choice points for the
third goal of the Prolog Query, and so on. At level $i$ of the tree, there are $N_u^{i-1}$ choice points. In order to deduce all of the answers to the Prolog query, the Prolog Machine must traverse this entire search space. For $M$ goals in a Prolog query, the entire search space consists of $N_u^M - 1$ choice points since:

$$N_u^0 + N_u^1 + N_u^2 + \ldots + N_u^{M-1} \leq N_u^M - 1$$

The complexity of the implicit Prolog control algorithm is $O(N_u^M)$. Thus, the Prolog control algorithm is exponentially complex. For each of the $N_u^M - 1$ choice points in the Prolog search space, a sequential database query is executed requiring the evaluation of $N$ database facts. Therefore, $NN_u^M - N$ unification attempts must be performed during the execution of our Prolog query. The number of unifiable database facts for each goal, $N_u$, is a subset of the $N$ database facts in the relation stack. Let $N_u = N/K$ where $K$ is a constant of proportionality describing the proportion of the relation stack containing unifiable database facts:

$$NN_u^M - N = (N^{M+1}/K^M) - N$$

The complexity is $O(N^{M+1})$. The complexity grows
exponentially as the number of database facts in a relation stack $N$ grows since $N_u$ grows according to the constant of proportionality $K$. This result is no surprise since we know that the Prolog control algorithm is of exponential complexity.

The purpose of this exercise was to determine the complexity of the Prolog Database System. Although the model is extremely simplified, the complexity of the Prolog Database System is obviously exponential due to the implicit control algorithm of Prolog. From the database management perspective, the best that we can do to save time and memory is to reduce unification attempts and page faults using indexing and efficient page buffering techniques. More complete research has been conducted on query complexity [4][26].
Chapter 6.0 The Look Ahead Page Replacement Policy

A page replacement policy is a set of rules applied to determine which database buffer page to select for replacement in the event of a page fault. Regardless of the page replacement policy used, there are bounds on the number of page faults that can occur during the satisfaction of a database query.

During the satisfaction of a sequential database query, all pages of a relation stack must be faulted into the database buffer so that unification attempts may be performed with all of the database facts on that relation stack. The lower bound on the number of page faults during the satisfaction of a sequential database query is therefore \( P \); the number of pages in a particular relation stack.

Due to the "one fact at a time" evaluation strategy of Prolog, it is possible that a database page could be faulted more than one time during the satisfaction of a database query. If a database page contains a unifiable fact, then that database page must also be accessed during the next instantiation of the database query since unevaluated facts remain on the page. A page fault will be generated for that page during the next instantiation of the database query if it is not still in the database buffer. Since this could happen for every unifiable database fact in the relation
stack, the upper bound on the number of page faults for a sequential database query is $P + N_u$; the number of pages on the relation stack plus the number of unifiable database facts on that relation stack.

When the goal of a database query contains bound arguments, indexing is used to retrieve only database pages with potentially useful database facts. The lower bound on page faults for an indexed database query occurs when indexing generates a reduced search space referencing only the pages in the relation stack that contain unifiable database facts, $P_u$. The upper bound on page faults for an indexed database query occurs when the reduced search space contains all $P$ of the relation stack pages and the page replacement policy does not retain those pages in the database buffer between instantiations of the database query.

**PAGE FAULT BOUNDARIES**

<table>
<thead>
<tr>
<th>Sequential Query</th>
<th>$P &lt; \text{page faults} &lt; P + N_u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed Query</td>
<td>$P_u &lt; \text{page faults} &lt; P + N_u$</td>
</tr>
</tbody>
</table>

The look ahead page replacement policy was designed for the Prolog Database System to approach the lower bound on page faults by retaining pages in the database buffer that
are most likely to be accessed again in the near future.

To satisfy a sequential database query, all of the database pages on a relation stack must be faulted into the database buffer. After a goal is unified with a fact on a database page, look ahead unification is performed on the database page to determine whether the next unifiable database fact for that goal is on the page. If that database page contains the next unifiable database fact, then the look ahead page replacement policy will attempt to retain that backtrackable page in the database buffer until Prolog backtracks to the goal. If this page is not retained in the database buffer, then it will be faulted back into the database buffer for a second time when Prolog backtracks to the goal. Retaining a working set of the most recently used database buffer pages that are backtrackable helps eliminate thrashing and minimize page faults.

The identification of non-backtrackable database pages also avoids potential page faults. The backtrack point for a database query is advanced to the base of the next page in the relation stack when the next unifiable database fact is not found on a page. A page replacement policy that does not do this would eventually backtrack to this database page only to find that the next unifiable database fact is not on the page. If this page were not still in the database buffer after backtracking, then an avoidable page fault
would have been generated. Looking ahead on database pages while they are in the database buffer avoids expensive page faults.

One potential advantage of look ahead processing is that page fault service times can be reduced if asynchronous I/O is available. If the next unifiable database fact is not in the database buffer, then the next page of the relation stack can be faulted into main memory in parallel with the resumed execution of the Prolog machine. The next page of the relation stack will then be in the database buffer after the Prolog machine backtracks to the goal and resumes the database query. The cost of such a page fault will be diminished considerably since the data transfer process is executed in parallel with the Prolog Machine. However, precautions must be taken to avoid accessing the data before the transfer is complete.

6.1 Look Ahead Unification

The purpose of look ahead unification is to determine the backtrack status of the database pages while they are in the main memory database buffer. If the next unifiable fact is not found on a database page during look ahead unification, then the backtrack point for that database query is advanced to the next page in that relation stack.
If that next page is in the database buffer, then look ahead unification proceeds on it, otherwise, the first fact on that page becomes the backtrack point for the database query. After look ahead unification, the backtrack pointer either points to the next unifiable database fact for that goal or to the next database fact in that relation stack that is not currently in the database buffer.

The database page containing a backtrack point is marked as backtrackable so that the look ahead page replacement policy will not select it as a replacement page. A backtrack count array records the backtrack status of all the database pages. The array contains one element for every database page where each element is an integer representing the number of database queries with a backtrack point on that page. The backtrack count for a database page is incremented when look ahead unification results in a backtrack point on that page. The backtrack count for a database page is decremented after Prolog backtracks to the goal for a database query that contains a backtrack point on that page, but only if the next backtrack point is not on that page.

The backtrack count array is wasteful of memory since it requires as many elements as there are database pages. However, space can be saved by limiting the size of each array element to just a few bits and access time is constant
for the backtrack count of any database page. Since the fact database is expected to be very large, a more memory efficient yet time consuming approach may be a dynamic backtrack count search tree containing a backtrack count node for each backtrackable database page. The trade-off for reduced memory usage is increased access time since binary search tree complexity is $O(\log_2 N)$ where $N$ is the number of backtrackable database pages.

Alternatively, it may not be necessary to maintain a backtrack count for every page in the database. Instead, a backtrack count could be maintained only for pages in the database buffer. However, this approach could degrade the performance of the page replacement policy. To see this, consider the situation where every page in the database buffer is backtrackable and a page fault occurs. A backtrackable page must be selected for replacement and the backtrack count for that page will be lost. When the backtrackable database page is later read back into the database buffer, the backtrack count for that page would have to be assumed to be zero, although other goals may consider that page as backtrackable.

Saving backtrack counts only for pages in the database buffer means that there may be backtrackable pages that the page replacement policy does not consider as backtrackable. The result is that the page replacement algorithm would only
be attempting to retain backtrackable pages that have remained in the database buffer since they were marked as backtrackable. In this version of the Prolog Database System, we maintain backtrack counts for every database page.

Another alternative is to store the backtrack count for a database page on that page. However, the possibility of a cut predicate causes problems with this approach. Backtrack counts for pages containing backtrack points of goals that are affected by the cut could only be decremented by accessing those database pages causing potential page faults. The backtrack count array offers efficient access to the backtrack counts of all database pages so that the integrity of the backtrack counts can be maintained.

6.1.1 Full Look Ahead Unification

After a goal is unified with a database fact, look ahead unification (Figure 10) is performed to determine the backtrack status of the current database buffer page. Before look ahead unification begins, the arguments of the goal must be restored to their original state. The Database Manager un-unifies the currently unified database fact from the goal saving the address of that fact for later re-unification after look ahead unification is complete.
procedure Find_Backtrack_Point
  un-unify the goal
  save the address of currently unifiable fact
  for current page .. top page of relation stack
    if page is not in database buffer then
      save next fact as backtrack point
      mark backtrackable page
      re-unify goal with currently unifiable fact
      return unified goal and backtrack point to Prolog
    end if
  for the rest of the facts on the current page
    attempt unification with the fact
    if successful unification then
      un-unify the goal
      save address of backtrack point
      mark backtrackable page
      re-unify the goal with the currently unifiable fact
      return unified goal and backtrack point to Prolog
    end if
  end for
end for
end Find_Backtrack_Point

Figure 10
Look Ahead Algorithm
Depending upon the outcome of look ahead unification, the goal will be in one of two possible states. The goal will be unified with the next unifiable database fact if look ahead unification is successful. In this case, the address of the fact is saved onto the Prolog Stack as the backtrack point for the database query and the database fact is un-unified from the goal so that the goal can be re-unified with the currently unifiable database fact and returned to Prolog.

If look ahead unification fails to locate the next unifiable database fact, then the goal arguments are in their original state. The goal argument is re-unified with the currently unifiable database fact and returned to Prolog along with the backtrack point. In this case the backtrack point is the next fact on the relation stack that is not in the database buffer.

Look ahead unification attempts to unify the goal with the same database facts with which unification would have been attempted during the next instantiation of the database query. The purpose of look ahead unification is to pre-perform unification while the database page is in the database buffer. There is, however, overhead associated with look ahead unification; un-unifying the currently unifiable database fact from the goal before look ahead unification and then re-unifying it after look ahead
unification is complete. This constant amount of overhead computation added by performing look ahead unification occurs $N_u$ times for every database query assuming that every goal unifies with $N_u$ database facts. Since the Prolog control algorithm is exponentially complex, this overhead could be significant unless the look ahead page replacement policy makes effective use of the backtrack information acquired.

6.2 The Abnormal Termination of a Database Query

After the Database Manager unifies the goal with a database fact and look ahead unification is performed, control is returned to Prolog and the database query is not active until Prolog backtracks to the corresponding goal. Up to this point, it has been assumed that Prolog will eventually backtrack to the goal for the next answer. In reality, this may not be the case.

A cut goal is used in a Prolog program to prune the search space and complexity of the Prolog control algorithm. By definition, a rule fails when Prolog backtracks across a cut goal within that rule. A cut has the effect of reducing the number of answers to the goals that precede the cut goal in the rule. Reducing the number of choices at choice points for these goals reduces the complexity of the search
space by pruning branches from the tree.

If Prolog backtracks across a cut goal within a rule containing a goal that generated a database query, then the rule will immediately fail and Prolog will not backtrack to the goal. If Prolog does not backtrack to the goal to decrement the backtrack count of the backtrackable page, then that backtrack count will not be accurate. For the duration of the Prolog query, that page will be considered as backtrackable, although it may not be backtrackable.

The solution is to force Prolog to backtrack to the goal for the sole purpose of decrementing the backtrack count of the formerly backtrackable page. The Prolog Machine will be slowed as it backtracks since it must decrement the backtrack count of a database page at every choice point corresponding to a Prolog goal containing a database predicate. However, this seems to be the best solution to this difficult problem.

If backtrack counts were maintained only for pages in the database buffer, then there might be fewer backtrack counts to decrement. However, the Prolog Machine would still have to stop at every choice point to determine whether the backtrackable page is in the database buffer. The problem of maintaining accurate backtrack counts only for pages in the database buffer was discussed in section 6.1.
Another example of the abnormal termination of a database query is the premature completion of a Prolog query. When Prolog finds an answer to the Prolog query during an interactive session, the user has the opportunity to request more answers. If the user decides not to pursue more answers, then the Prolog query is complete although active database queries may still exist. The backtrack counts for the backtrackable pages of these database queries must be reset to zero before the next Prolog query is issued. The solution is to reset the backtrack counts of all backtrackable pages.

Look ahead unification is performed so that the page replacement policy can attempt to retain database buffer pages that are backtrackable. The results of look ahead unification are wasted when a database query is terminated abnormally since Prolog never backtracks to the backtrackable page. Unfortunately, there is no way to determine ahead of time that a database query is going to be terminated abnormally.

One possible solution to this problem is to maintain profiles from other runs and to use the statistics to determine when not to perform look ahead unification. For instance, historical results may indicate that certain goals or combinations of goals tend to be prone to abnormal termination. However, an algorithm that decides whether to
perform look ahead unification could be difficult to construct and adapt in a continuously changing environment.

6.3 The Page Replacement Policy

A page replacement policy is an ordered set of rules used to select a replaceable database buffer page in the event of a page fault. The look ahead page replacement policy states that the least recently used database buffer page that is not backtrackable is selected as the replacement page. If all of the database buffer pages are backtrackable, then the least recently used one is selected. The look ahead page replacement policy attempts to retain a working set of the most recently used database buffer pages that are backtrackable.

It is possible that multiple database buffer pages fall into either the backtrackable or non-backtrackable page categories. For example, there may be multiple database buffer pages that are backtrackable. The selection of a replacement page is determined by choosing the least recently used database buffer page that meets the criterion of the category.

The most recently used backtrackable database buffer page would be the next page referenced if Prolog were to start backtracking immediately. It is important to retain
backtrackable database buffer pages that correspond to the Prolog goals closest to the current point of Prolog execution. Locality of reference is inherent in Prolog due to the stack nature of backtracking. For this reason, the look ahead page replacement policy was designed to retain a working set of the most recently used database buffer pages that are backtrackable.

Prolog may repetitively evaluate a local set of goals due to backtracking and goals with multiple answers. The most recently used database buffer pages that are not backtrackable contain database facts with the same database predicates as the Prolog goals closest to the current point of Prolog execution. It is possible that potential page faults may be avoided by retaining the most recently used database buffer pages that are not backtrackable.

6.3.1 The Page Replacement Policy Rules

The purpose of this section is to describe the look ahead page replacement rules in a concise manner. Although the page replacement rules were implemented in the C programming language, they are presented here using Prolog to promote a concise description at a symbolic level of abstraction. Before this can be accomplished, a Prolog representation of the backtrack count data structure and the
page table must be defined.

For every backtrackable database page, there exists a fact identifying the backtrack count for that page. The first argument identifies the backtrackable database page. The second argument identifies the backtrack count for that page:

\[(\text{backtrackable } \text{Page } \text{Backtrack\_count})\]

Only backtrackable database pages are represented. The lack of a backtrack count fact for a given database page indicates that the page is not backtrackable. A query is issued to determine the backtrack count of a given database page:

\[(\text{backtrackable } ?\text{page } ?\text{count})\]

Assuming that the Prolog variable ?page is initially bound to a page identifier, the query will fail if the page is not backtrackable. If the database page is backtrackable, the query will succeed binding the Prolog variable ?count to the backtrack count for that page.

The accuracy of the backtrack count data structure is maintained when a backtrack point is found on a database page. A backtrack count of one is asserted for a database
page that was not previously backtrackable:

( mark_page ?page ) if
   ( not ( backtrackable ?page ?count ) )
   ( assert ( ( backtrackable ?page 1 ) ) )
   ( cut )

The backtrack count of a previously backtrackable database page is incremented when a backtrack point is found on that page:

( mark_page ?page ) if
   ( backtrackable ?page ?count )
   ( = ?new_count ( + ?count 1 ) )
   ( asserta ( ( backtrackable ?page ?new_count ) ) )
   ( cut )

The page table is represented as a set of Prolog facts identifying the database pages that are resident in the database buffer frames. For each database buffer frame, there exists a fact identifying the database page that currently resides in that frame. The first argument identifies the database page. The second argument identifies the database buffer frame where that page currently resides:

( is_in Page Frame )
For each database buffer frame, there also exists a fact indicating the status of the database page in that particular database buffer frame. The first argument identifies the database buffer frame. The second argument indicates whether the database page in that frame is clean or dirty:

( status Frame clean ) or ( status Frame dirty )

A page request is issued when a database fact on that page is requested for a unification attempt:

( page_request ?requested_page ?frame ) if
  ( is_in ?requested_page ?frame )
  ( retract ( ( is_in ?requested_page ?frame ) ) )
  ( assertz ( ( is_in ?requested_page ?frame ) ) )
  ( cut )

Assuming the Prolog variable ?requested_page is initially bound to a page identifier, the query will succeed if the requested page is currently in the database buffer. The Prolog variable ?frame will be bound to the identifier for the database buffer frame where the page resides.

The page table fact for that frame will be retracted and asserted to the end of the Prolog fact and rule list.
indicating that the page in that frame is the most recently used page in the database buffer. Prolog will evaluate the page table facts in least recently used order when the most recently used fact is asserted at the end of the Prolog fact and rule list.

A page fault is generated when the requested page is not found in the database buffer. The first rule of the look ahead page replacement policy states that the least recently used database buffer page that is not backtrackable should be selected as the replacement page:

( page_request ?requested_page ?replace_frame ) if
( is_in ?replace_pg ?replace_frame )
( not ( backtrackable ?replace_pg ?count ) )
( replace ?replace_pg ?replace_frame ?requested_page )
( cut )

The page table facts will be traversed in least recently used order until a database buffer page is found that is not backtrackable. The replacement page is replaced with the requested page. The Prolog variable ?replace_frame will be bound to the identifier for the database buffer frame where the requested page resides.

If the first page fault rule fails, then the database buffer is full of backtrackable pages. The final page fault rule states that the least recently used database buffer
page that is backtrackable should be selected as the replacement page:

\[(\text{page}_\text{request} \ ?\text{requested}_\text{page} \ ?\text{replace}_\text{frame}) \text{ if} \]
\[(\text{is}_\text{in} \ ?\text{replace}_\text{pg} \ ?\text{replace}_\text{frame}) \]
\[(\text{replace} \ ?\text{replace}_\text{pg} \ ?\text{replace}_\text{frame} \ ?\text{requested}_\text{page}) \]
\[(\text{cut})\]

The replacement of the replacement page with the requested page is accomplished by first determining whether the replacement page is clean or dirty. The requested page can be written over a clean replacement page:

\[(\text{replace} \ ?\text{replace}_\text{pg} _?\text{replace}_\text{frame} \ ?\text{requested}_\text{page}) \text{ if} \]
\[(\text{status} \ ?\text{replace}_\text{frame} \text{ clean}) \]
\[(\text{read} \ ?\text{requested}_\text{page} \text{ into} \ ?\text{replace}_\text{frame}) \]
\[(\text{retract} \ ((\text{is}_\text{in} \ ?\text{replace}_\text{pg} \ ?\text{replace}_\text{frame}) \ )) \]
\[(\text{assertz} \ ((\text{is}_\text{in} \ ?\text{requested}_\text{page} \ ?\text{replace}_\text{frame}) \ )) \]
\[(\text{cut})\]

A dirty replacement page must be written to disk before the requested page can be read into the replacement frame:
( replace ?replace_pg ?replace_frame ?requested_page ) if
( status ?replace_frame dirty )
( write ?replace_pg from ?replace_frame )
( read ?requested_page into ?replace_frame )
( retract ( ( is_in ?replace_pg ?replace_frame ) ) )
( assertz ( ( is_in ?requested_page ?replace_frame ) ) )
( retract ( ( status ?replace_frame dirty ) ) )
( assert ( ( status ?replace_frame clean ) ) )

6.4 Look Ahead Unification and Indexing

When a Prolog goal contains an argument containing an atom, the Index Manager uses the perfect hashing technique [18] to generate a reduced search space consisting of the addresses of database facts that will unify with the corresponding argument of the goal. Reducing the search space eliminates from consideration some of the database facts that will not unify with the goal reducing the number of unification attempts necessary to satisfy the database query.

Another advantage of indexing is that only database pages containing possibly unifiable database facts must be faulted into the database buffer when the facts in the reduced search space are traversed. Every database page of a relation stack page chain must be faulted into the database buffer when an entire relation stack is traversed sequentially. Also, indexed look ahead unification may
proceed further in the relation stack than sequential look ahead unification. Sequential look ahead unification may halt due to a database page not in the database buffer although that page does not contain any unifiable database facts. Indexed look ahead unification will not need to access that page when it is not referenced in the reduced search space. Indexed look ahead unification can proceed on the next database page of the relation stack page chain that is in the database buffer and is referenced in the reduced search space.
Chapter 7.0 Experimental Design and Setup

To determine the effectiveness and efficiency of the look ahead page replacement policy, thirty sample Prolog queries were executed using a sample fact database. The experiment was conducted for both the least recently used page replacement policy (LRU PRP) and the look ahead page replacement policy (LA PRP). The results are presented in the next chapter.

The LRU PRP was chosen for comparison since it is the basis of the LA PRP. The LRU PRP selects the least recently used database buffer page as the replacement page in the event of a page fault. The design of the LRU PRP is based on the locality of Prolog execution as is the LA PRP. The history of database buffer page usage becomes a secondary criterion with the LA PRP. For the LA PRP, the backtrack status of the database buffer pages is the primary criterion used in the selection of a replacement page. The LA PRP gives a high priority to retaining backtrackable database buffer pages. The selection of a replacement page becomes more complex due to this additional criterion.

7.1 The Break Even Analysis Model

The major cost of the LA PRP is look ahead unification.
Every time a goal is unified with a database fact, look ahead unification is performed to determine the backtrack status of the database buffer pages. The number of unification attempts performed determines the magnitude of the cost. If the costs are averaged over the number of look ahead unification attempts (Avg LA Cycle Time), then cost can be expressed as:

\[
\text{Cost(LA)} = \text{LA Cycles} \times \text{Avg LA Cycle Time}
\]

The benefit of the LA PRP is execution time saved due to a reduction in page faults. The benefit is expressed by multiplying the average-page fault time (Avg PgFlt Time) by the number of page faults saved (PgFltsSvd):

\[
\text{Benefit(LA)} = \text{PgFltsSvd} \times \text{Avg PgFlt Time}
\]

where: \(\text{PgFltsSvd} = \text{PgFlts(LRU)} - \text{PgFlts(LA)}\)

then: \(\text{Time(LRU)} - \text{Time(LA)} = \text{Benefit(LA)} - \text{Cost(LA)}\)

When benefits exceed costs, the LA PRP outperforms the LRU PRP with respect to execution time. When costs are equal to benefits, a break even point is reached where the LA PRP and the LRU PRP result in identical execution times:
\[
\text{Time}(\text{LRU}) - \text{Time}(\text{LA}) = 0
\]

when: \(\text{Benefit}(\text{LA}) = \text{Cost}(\text{LA})\)

The break even equation is expressed in greater detail by expanding the benefit and cost terms as follows:

\[
\text{Benefit}(\text{LA}) = \text{Cost}(\text{LA})
\]

\[
\text{PgFltsSvd} \times \text{Avg PgFlt Time} = \text{LA Cycles} \times \text{Avg LA Cycle Time}
\]

\[
\text{LA Cycles} / \text{PgFltsSvd} = \text{Avg PgFlt Time} / \text{Avg LA Cycle Time}
\]

The break even point is reached when the number of look ahead unification attempts necessary to save one page fault is equal to the number of look ahead unification attempts that can be executed in the time it takes for one page fault to execute. The number of look ahead unification attempts necessary to save one page fault is referred to as the look ahead effectiveness ratio. The next chapter includes look ahead effectiveness ratio results for a sample set of Prolog queries and a sample fact database.

### 7.2 The Sample Fact Database

Determining the effectiveness and efficiency of the LA PRP requires a sample set of Prolog database queries to be executed on a sample Prolog fact database. A database
provided by the Naval Surface Warfare Center (NAVSWC) was considered for this purpose. However, the database was determined to be inadequate since it did not possess the characteristics (as discussed below) of a fact database that the Prolog Database System is expected to query.

Although the NAVSWC database contained an adequate number of relations with adequate populations, the database was incomplete. The data provided were the unclassified portion of a partially classified database. Since the data were only a small and arbitrary subset of the total database, they were inadequate providing incomplete connectivity between relations and an inadequate variety of atoms within the relations.

The connectivity between the relations of the database suffered because many of the atoms were only repeated in the classified portion of the database which was not supplied. Connectivity also suffered because the remainder of the atoms were repeated many times in a particular relation argument, thus there was a poor variety of unique atoms in the relations. Poor connectivity and variety made it extremely difficult to construct complex queries that would exercise the LA PRP adequately.

To compound the connectivity and variety problems, all of the repeated atoms in the first argument position were grouped together. Groupings of repeated atoms were also
found in many other relation arguments. The NAVSWC database was not representative of the fact databases that the Prolog Database System is expected to encounter since the fact databases built and maintained by the Prolog Database System are not expected to have the facts of the relations sorted on the first argument.

Another problem with the NAVSWC database was that all of the relations contained only one or two argument positions. A more realistic database would include relations with higher order arity. For these reasons of connectivity, variety, and arity, the incomplete NAVSWC database was not used for experimentation purposes.

Instead of using an existing database with all of its application specific dependencies, a database was generated (Figure 11) that is intended to be representative of databases that the Prolog Database System is expected to encounter. The database generated provides connectivity between relations with a large range of arities and a good variety of atoms that the incomplete NAVSWC database did not provide. Due to the exponential complexity of the system, the number of facts generated for each relation was small in order to constrain query execution times.

The program generates three sets of relations with some differing characteristics. All three sets contain relations with differing arities ranging from two to five arguments.
main () {

    /*
     * relation args density facts */

    makedb ( "fiveSmall.db", 5, 5, 2500 );
    makedb ( "fourSmall.db", 4, 5, 2500 );
    makedb ( "threeSmall.db", 3, 5, 2500 );
    makedb ( "twoSmall.db", 2, 5, 2500 );
    makedb ( "five.Dense.db", 5, 8, 5000 );
    makedb ( "four.Dense.db", 4, 8, 5000 );
    makedb ( "three.Dense.db", 3, 8, 5000 );
    makedb ( "two.Dense.db", 2, 8, 5000 );
    makedb ( "five.db", 5, 13, 5000 );
    makedb ( "four.db", 4, 13, 5000 );
    makedb ( "three.db", 3, 13, 5000 );
    makedb ( "two.db", 2, 13, 5000 );

} /* main */

void makedb ( relation, args, density, facts )

    char *relation;
    int args, density, facts;

    {
        int i, j, k, fd;
        char letter;

        fd = open ( relation, O_RDWR );
        for ( i=0; i < facts; i++ ) { /* write a fact */
            for ( j=0; j < args; j++ ) { /* write an argument */
                for ( k=0; k < 2; k++ ) { /* write a letter */
                    letter = ( random() % density ) + 65;
                    write ( fd, &letter, sizeof(letter) );
                    write ( fd, " ", 1 );
                }
            }
            write ( fd, 

        close ( fd );

} /* makedb */

Figure 11

Database Generation Program
Two of the sets contain relations with five thousand database facts each while the third set contains relations with two thousand five hundred database facts each. The database generated contains relations with a wide range of arities and varying populations.

The basic assumption in generating this database is that some atoms will occur more frequently than others. To achieve this effect, the first set of relations was constructed from a basic set of atoms. The second relation set was constructed from a subset of this basic set and the third relation set was constructed from a subset of the atoms used in the second set.

The following table summarizes the distribution of atoms throughout a database generated with the program of Figure 11.

<table>
<thead>
<tr>
<th>Atom Subset</th>
<th># of Atoms</th>
<th>Frequency (F)</th>
<th>Rank (R)</th>
<th>F * R</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>25</td>
<td>2900</td>
<td>1</td>
<td>2900</td>
</tr>
<tr>
<td>#2</td>
<td>39</td>
<td>1500</td>
<td>2</td>
<td>3000</td>
</tr>
<tr>
<td>#3</td>
<td>105</td>
<td>425</td>
<td>3</td>
<td>1275</td>
</tr>
</tbody>
</table>

There are three mutually exclusive subsets of all atoms generated in the program. The first subset refers to the atoms generated for the first four relations in the main section of the program where density is equal to five. Since there are five possible characters that can be
generated and two characters are concatenated to form an atom, there are 25 possible atoms in that subset. The second subset refers to the atoms generated in the next four relations of the program where density is equal to eight. There are 64 possible atoms generated, 25 of which are included in the first subset leaving 39 atoms that are found in those relations but not in the first four relations. The number of atoms in the third subset are calculated similarly. There are 105 atoms found in the last four relations of the program that are not found in any of the other relations.

Analysis of a database generated with the program shows that atom frequency multiplied by rank is close to constant for the first two sets of atoms much like English text as indicated by Zipf's Law [22]. However, the atoms are more densely distributed than words in English text much like atoms would be distributed in a database not consisting of English text. Also the third set of atoms is much less densely distributed than the other two sets much like the atoms in a fact database where a small subset of the atoms occur much more frequently than the others.

Although the atoms have no symbolic meaning, the high frequency of atom occurrence and the large variety of atoms ensure that all of the relations have atoms in common providing good connectivity between relations. The
following Prolog facts are examples of those generated by the program of Figure 11 for the "fivedense" relation:

```
fivedense( AH DG AE ED EC )
fivedense( HD AH GD CD DH )
fivedense( GB HF FH DH CE )
fivedense( GC EB BE FF HB )
fivedense( HG FH FD CH GF )
fivedense( GF HF CE FF DH )
fivedense( CB BG DC CA HC )
```

Connectivity and a large variety of atoms provide a database that is needed to generate a good sample set of queries to exercise the Prolog Database System.

7.3 The Sample Prolog Queries

Thirty sample Prolog queries were generated to test the two page replacement policies. The queries were designed to exercise the look ahead page replacement code thoroughly. Some of the sample Prolog queries contain goals with all unbound variables. This ensures that sequential database queries will occur. Some goal arguments are bound to a particular atom to ensure that indexed database queries will occur. The following are two of the sample Prolog queries used in the experiment:
( query1 ) if
  ( threeDense AA AB ?a )
  ( twoDense AA ?b )
  ( fourDense ?a ?b ?c ?d )
  ( four ?a AA ?x ?y )
  ( three AA ?n ?m )
  ( two AA ?o )
  ( three AA AA ?p )
  ( fourSmall ?a ?b ?c ?d )
  ( fail )

( query2 ) if
  ( threeSmall ?a ?a ?a )
  ( fiveSmall ?a ?b ?a ?d ?a )
  ( four ?b ?e ?f ?g )
  ( threeDense ?b ?y ?a )
  ( threeSmall ?y ?x ?x )
  ( fiveSmall ?x ?x ?a ?j ?k )
  ( two ?d ?k )
  ( fiveSmall ?a ?c ?f ?i ?x )
  ( fail )

The thirty sample Prolog queries were designed to represent a wide range of query complexities. Table 1 provides some statistics on the queries when the database buffer consists of four frames that are each eight kilobytes in size. Listed for each query are the number of page faults generated when that query is run on the sample database, the number of solutions to the query, the number of goals and unique variables in that query, as well as the number of relations accessed by the query. Figure 12 presents the same page fault statistics, however, they are arranged in a graphical format to illustrate the range of page faults generated by the queries.
Table 1

Query Statistics

<table>
<thead>
<tr>
<th>Query</th>
<th>Pg Flts</th>
<th>Solns</th>
<th>Goals</th>
<th>Vars</th>
<th>Rels</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>503</td>
<td>28</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>902</td>
<td>100</td>
<td>4</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2395</td>
<td>120</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3464</td>
<td>84</td>
<td>4</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>19841</td>
<td>896</td>
<td>4</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>137</td>
<td>59</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7227</td>
<td>436</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>6893</td>
<td>299</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>463</td>
<td>120</td>
<td>3</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>4623</td>
<td>248</td>
<td>3</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>1282</td>
<td>58</td>
<td>3</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>6788</td>
<td>1746</td>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>405</td>
<td>28</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>1672</td>
<td>47</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>207</td>
<td>9</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>27303</td>
<td>916</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>274</td>
<td>13</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>16526</td>
<td>1036</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>1873</td>
<td>78</td>
<td>3</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>3298</td>
<td>79</td>
<td>4</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>4328</td>
<td>232</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>165</td>
<td>31</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>8838</td>
<td>310</td>
<td>8</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>25</td>
<td>8899</td>
<td>437</td>
<td>8</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>35965</td>
<td>1642</td>
<td>6</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>27</td>
<td>5183</td>
<td>258</td>
<td>11</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>28</td>
<td>49380</td>
<td>6248</td>
<td>7</td>
<td>12</td>
<td>7</td>
</tr>
<tr>
<td>29</td>
<td>60333</td>
<td>2873</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>30</td>
<td>39914</td>
<td>1364</td>
<td>7</td>
<td>13</td>
<td>5</td>
</tr>
</tbody>
</table>
Figure 12
Representative Page Fault Range
Chapter 8.0 Empirical Results

A performance comparison was conducted for the LRU and LA page replacement policies by executing the sample Prolog queries under various combinations of Database configurations. The tests were conducted with database page sizes of two, four, and eight kilobytes. Database buffer size was varied from four to eight to sixteen frames. Each database query was executed with the Database configured in one of these nine possible configurations. For each configuration of the Database Module, the database queries were executed using the LRU PRP and the LA PRP.

These tests were conducted on a Gould Power Node Model 6042 running the UTX/32 derivation of the Unix System V operating system. Main memory is implemented as twelve megabytes of physical memory extended to sixteen megabytes of virtual memory. This machine is capable of performing .972 million Whetstone instructions per second with no other users on the system.

Execution times for the sample Prolog queries were measured using the system wall clock. This was necessary since the system cpu clock measures only cpu time. The system wall clock is the only instrumentation available that can be used to measure total query execution time including both cpu time and page transfer time. To ensure that the
sample Prolog queries were run under similar circumstances for both page replacement policies, all unnecessary UNIX daemons were turned off and all terminals were turned off so that no other users could log onto the system.

8.1 General Results

Figures 13 and 14 show the average execution time per sample Prolog query while figures 15 and 16 show the average number of page faults per sample query. These figures indicate that the LA PRP is superior to the LRU PRP for this experiment, however, averages alone can be deceiving. The sign test confirms that the improvement was consistent across the sample queries for each database configuration by yielding a P-value of zero (to three decimal places) for all database configurations with respect to both time and page faults.

These figures also indicate that execution time and page faults decreased as the page size increased and decreased only slightly as the number of database buffer frames increased. More database facts can be packed onto a larger page decreasing the number of database pages in the fact database. Obviously, fewer page faults will occur as the number of database pages in the fact database decreases independently of the page replacement policy used.
Figure 13

Execution Time Relative to Page Size
Figure 14

Execution Time Relative to Buffer Size
Figure 15
Page Faults Relative to Page Size
Figure 16
Page Faults Relative to Buffer Size
More database buffer frames decrease page faults by increasing the chances that a requested page will be found in the database buffer. A significant increase in database buffer size is required to affect this probability when the fact database contains thousands of database pages.

Figure 17 illustrates page fault times for the various database page sizes. Although page fault times increase with page size, execution times manage to decrease due to the more prominent decrease in page faults.

8.1.1 Page Fault Time Measurement

Since query execution times were measured using the system wall clock, there was no way to determine how much of that time was attributable to page faults. As previously mentioned, page faults require cpu time to set up the data transfer and non-cpu time to transfer the data. To acquire a measure of page fault time for a query, the sequence of pages faulted into the database buffer was recorded and a program was run that executed that sequence of page faults. Again, the system wall clock was used to measure the time necessary to execute those page faults so that the average time per page fault could be measured. Page fault times were acquired in an environment isolated from the execution of the Prolog Database System.
Figure 17
Average Page Fault Time
The fact database was stored and accessed through a raw disk interface bypassing the UNIX file system. The UNIX file system incorporates a block buffer cache that buffers the database buffer from the disk. In the event of a page fault, the block buffer cache prevents disk access when the requested page is resident in the block buffer cache. The raw disk interface is used instead to ensure that a page fault results in a disk access.

8.2 Margin of Improvement Analysis

Figures 18 and 19 illustrate the average margin of execution time improvement and page fault reduction for the LA PRP with respect to the LRU PRP. Figures 28 and 29 in the next chapter show that variance about the mean tends to increase with page and buffer size, however, variance should be interpreted with care since the population does not tend to be normally distributed.

The margin of improvement changes only slightly for the different database buffer sizes while rising significantly with page size. Larger database pages result in more facts per page increasing the probability that look ahead unification will succeed in finding the next unifiable database fact for a Prolog goal. Figure 20 verifies this observation.
Figure 18

Time Savings
Figure 19
Page Fault Savings
Figure 20

Look Ahead Success Relative to Page Size
For a smaller page size, look ahead is less likely to find the next unifiable database fact. Even in this event, look ahead advances the backtrack pointer past facts that will not unify. Page faults may be saved by advancing the backtrack pointer to the next page of the relation stack so that the database query will not resume on the current page. Without look ahead, the current page would be the first page accessed when the database query resumed only to find that the next unifiable database fact is not there. Look ahead examines the page while it is in the database buffer so that future page faults may be avoided. Database buffer pages that are not found to be backtrackable should be selected as replacement pages since they are of no future use in the satisfaction of the database query.

For a larger page size, look ahead is more likely to find the next unifiable database fact. The increasing success in look ahead unification seems to be related to the increasing margin in page fault reduction. This tends to indicate that the retention of backtrackable database buffer pages saves page faults more effectively than discarding pages that are not found to be backtrackable.

Figure 21 indicates that look ahead success is less dependent on database buffer size. The steady margin of page fault reduction and execution time improvements with respect to database buffer size verify this observation.
Figure 21
Look Ahead Success Relative to Buffer Size
In section 7.1.1, the look ahead effectiveness ratio (LAE) was defined as the number of look ahead unification attempts necessary to save a page fault. Figure 22 shows that for the sample queries, the LAE ratio tends to increase with increasing page size and database buffer size. As the page size and database buffer sizes increase, the number of database facts examined to determine that a page is not backtrackable increases. The number of database facts examined to find the next unifiable database fact is also likely to increase with page and database buffer sizes.

8.3 Results Summary

Figures 23 and 24 present overall averages supporting the observation that page size is the more prominent factor in determining the effectiveness of the LA PRP relative to the LRU PRP. Figure 25 illustrates that, overall, the LA PRP averaged a 3.8 percent reduction in page faults per query relative to the LRU PRP with a variance of .0024. Since the distribution does not tend to be normal, variance should be interpreted appropriately. The margin of execution time improvement per query averaged 2.9 percent with a variance of .0017. On average, the benefits of look ahead unification outweigh the costs for the given fact database and queries.
Figure 22
Look Ahead Effectiveness Ratio
Figure 23

Results Relative to Page Size
Figure 24
Results Relative to Buffer Size
Figure 25

Overall Results
These results indicate that the LA PRP is more effective, in this environment, at saving page faults than the LRU PRP and more efficient with respect to time than the LRU PRP. The two to three percent savings rates, however, are only considered as moderate improvements.

Future advancements in state of the art computing hardware and information storage devices will have an impact on the performance of the LA PRP. Advances in parallel processing will have positive impacts on the look ahead process. SIMD array processing will make it possible to perform look ahead unification on all of the facts in the database buffer in parallel. The time necessary to perform look ahead unification will approach the time necessary to perform just one look ahead unification attempt. If the overhead necessary to implement this parallel approach is not excessive, then it is a viable strategy.

As memory becomes less expensive and more available, the size of the database buffer will increase. Results presented here show that this memory should be allocated in the form of larger page sizes. Figure 18 indicates that larger page sizes increase the likelihood that look ahead unification will be successful resulting in greater page fault savings.

Page fault time will benefit less from the effects of faster processors and memories. Although these factors will
decrease the time necessary to activate device drivers and perform DMA data transfers, the physical movement of the magnetic and optical transducers will continue to be the limiting factor.

As these future technologies evolve, the look ahead overhead necessary to save a page fault will decrease more significantly than page fault time. The less expensive LA PRP will become more attractive as a means to avoid expensive page faults. The advantages of the LA PRP will grow into the future.

Future advancements are also possible in the development of better look ahead approaches. The next chapter outlines such an approach called partial look ahead unification. As in any technology, trade-offs are expected and partial look ahead unification is no exception as it trades off look ahead accuracy for look ahead efficiency. This less accurate unification algorithm is described in the next chapter.
Chapter 9.0  Summary and Concluding Remarks

Look ahead unification is performed to determine the backtrack status of a database page. The look ahead page replacement policy uses this information to prevent page faults. Although the experimental results indicate that full look ahead unification is effective, it is also costly since it involves both attempted unification and undoing a successful look ahead unification. It would be beneficial if the overhead associated with look ahead unification could be reduced while maintaining as much unification accuracy as possible. Although fewer page faults would be saved, the computation necessary to save those page faults would be decreased. If look ahead costs can be reduced more than benefits, then a look ahead algorithm exists that is more desirable than full look ahead unification. Partial Look Ahead Matching is one way of trading off look ahead accuracy for look ahead overhead.

9.1 Partial Look Ahead Matching

During the development and implementation of the look ahead page replacement algorithm, it was clear that look ahead unification was expensive and that less optimal approaches existed. As an alternative to full look ahead
unification, consider partial look ahead matching. The purpose of partial look ahead matching is to avoid the expensive unification overhead associated with full look ahead unification while retaining as much full look ahead unification accuracy as possible. Instead of using the Prolog argument registers for look ahead unification, the Page Manager constructs partial copies of the argument registers before they are modified by the unification process. The partial copies of the original argument registers are used for look ahead matching.

The partial copies of the argument registers are constructed by dereferencing the original argument registers. For example, an argument register containing a variable that is bound to an atom may be represented as a reference to a variable term that references an atom term. Dereferencing removes these references so that the value of the atom may be copied to the corresponding partial argument register. This approach reduces full look ahead unification to a type and value matching process. A database fact will partially match a goal if the types and values of the partial argument registers match the types and values of the corresponding database fact arguments.

The partial copy process could be designed so that only atomic term values such as atoms, strings, and numbers are copied to the partial argument registers. For instance, a
list value would not be copied to a partial argument register in order to minimize copy overhead and partial matching overhead. Also, no effort would be made to distinguish between two variables in the argument registers that reference the same variable. Only unification of types could be performed on lists and variables if this were the case.

The drawback is less accurate look ahead processing since partial look ahead matching could result in false backtrack points. Partial look ahead matching could, however, eliminate some of the database facts that will not unify with the goal. Although partial look ahead matching is not as accurate as full look ahead unification, it may reduce page faults more efficiently through a reduction in overhead per saved page fault.

9.1.1 Preliminary Results

Preliminary results confirm that partial look ahead (PLA) matching is less accurate than full look ahead (FLA) unification. PLA matching is more likely to find false backtrack points as page size increases resulting in the retention of database pages that may not be backtrackable. Although PLA matching results in fewer saved page faults than FLA unification, time savings are very similar (Figures
Although fewer large page faults were saved, the resulting time savings for large page faults are more significant than the time savings for smaller page faults. This coupled with the reduction in look ahead overhead resulting from PLA matching overcomes the reduction in look ahead accuracy.

The results for PLA matching are less stable than those for FLA unification, especially for a large page size. Figures 28 and 29 show that the results for the sample Prolog queries using FLA unification varied less about the sample mean than did those using PLA matching. The point estimate of the mean for the PLA matching population is not likely to be as accurate as the point estimate of the mean for the FLA unification population.

Overall, preliminary results indicate that PLA matching and FLA unification perform equally well with respect to time although PLA matching results in fewer saved page faults per sample query (Figure 30). A greater reduction in look ahead overhead should be realized when the capability to store lists and variables in the fact database is implemented. Future research in this area is necessary to accurately access the viability of PLA matching.
Figure 26
Page Fault Savings Averages
Figure 27

Time Savings Averages
Figure 28

Page Fault Savings Variances
Figure 29

Time Savings Variances
Figure 30

Preliminary Results
9.2 Project Summary

In the past, coupling Prolog with an existing DBMS has been the approach taken toward the extension of the fact storage and retrieval facilities of Prolog. The integration of a database module into Prolog requires more effort. A database module must be developed and integrated into Prolog. Jorge Bocca has stated, "As for systems which have attempted integration, they hardly exist" [2].

To satisfy the need for an integrated Prolog Database System capable of storing and retrieving a large number of facts, a custom database module was designed, developed, and integrated into Prolog. The resulting Prolog Database System is a single process, and provides efficiencies that a coupled multi-process Prolog Database System cannot provide:

1. Sharing of the Prolog Stack at choice points.
2. Sharing of the compact Prolog atom representation.
3. Fact storage tailored to the needs of Prolog.
4. The look ahead page replacement policy.

Integrating a custom built database module into Prolog affords the designer the flexibility to take advantage of
existing Prolog data structures and execution characteristics. The efficiency gain is realized in terms of response time and reduced operating system resource usage.

The research presented in this paper is intended to advance the understanding of database techniques that serve to extend the Prolog fact base as efficiently as possible. It is due to these efficiencies that an integrated Prolog Database System will be desirable in the future as confirmed by Jarke and Vassiliou:

"Going a step further, we argue that future decision support systems will have to integrate all three components: database, mathematical subsystem, and knowledge base" [13].

In summary, this project has contributed to the field of computer science one particular approach to the design of a database module integrated into Prolog. Specific contributions within the database module include a storage scheme for database facts that allows random access to any argument of a fact without traversing any of the other arguments. The most significant contribution is the look ahead page replacement policy. Although the experimental results show only a moderate 3.8% performance improvement over the least recently used page replacement policy, the results are consistently positive over all queries.
Bibliography


Vita

The author, J. Steven Haugh, was born on September 6, 1962 in Harrisonburg, Virginia. An undergraduate degree in computer science was received in 1984 from James Madison University in the city of birth. Employment since 1984 has been with the Naval Surface Warfare Center (NAVSWC) in Dahlgren, Virginia. Research conducted at NAVSWC includes the Prolog Database System and the NAVSWC Reactive Strategic Planner. Prolog, Object Oriented Pascal, C, and ADA are the programming languages used for prototype and implementation purposes. This project is submitted for partial fulfillment of the requirements for a Master's Degree in Computer Science from Virginia Polytechnic Institute and State University located in Blacksburg, Virginia. Future plans include the pursuit of a Ph.D in Computer Science from the same school and the continued research and development of the NAVSWC Reactive Strategic Planner.

Signature ____________________________

J. Steven Haugh