THE DESIGN, IMPLEMENTATION, AND PERFORMANCE EVALUATION OF BERMUDA

by

Yannis E. Ioannidis and Manolis M. Tsangaris

Computer Sciences Technical Report #973

October 1990

THE DESIGN, IMPLEMENTATION, AND PERFORMANCE EVALUATION OF BERMUDA

Yannis E. Ioannidis
Manolis M. Tsangaris

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

(Submitted for publication)

THE DESIGN, IMPLEMENTATION, AND PERFORMANCE EVALUATION OF BERMUDA [1,2]

Yannis E. Ioannidis
Manolis M. Tsangaris [3]

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

We describe the design and implementation of **BERMUDA**, which is a loosely-coupled system interfacing Prolog to the Britton-Lee Intelligent Database Machine (IDM-500). BERMUDA allows multiple concurrent Prolog processes, possibly running on different machines, to share a database. In addition, it preserves the semantics of Prolog programs and makes the use of the database system transparent to the user. We discuss several architectural issues faced by such systems and the approach adopted for each one in BERMUDA. We also present the performance results of a variety of experiments with the system. These include single-user benchmarks of BERMUDA against stand-alone Prolog and stand-alone IDM, detailed profiling of the costs of the modules of BERMUDA that shows the overhead imposed by the interface, and multi-user benchmarks of BERMUDA that show how the system behaves under heavier load. These experiments demonstrate the effectiveness of loosely-coupled systems in general and of BERMUDA specifically. They also help in the identification of some aspects of the design and implementation of BERMUDA that could be improved.

1. INTRODUCTION

Despite their potential performance inferiority, loosely-coupled systems are expected to play a significant role in future computational systems. The rate at which user demands on system functionality grow is usually higher than the rate at which integrated systems offering the desired services are developed. On the other hand, quite often, these services are collectively offered by a group of systems, which if put together would satisfy the user demands. Development time of such loosely-coupled systems is usually short and the resulting products have the potential of providing adequate functionality and performance. Moreover, there is always the need to integrate existing heterogeneous systems into a cohesive environment without sacrificing the privacy and/or independence of the participating systems. Hence, loosely-coupled systems are expected to continue to play a significant role. In addition, there are several important aspects of such systems that are not well understood, and whose study will help clarify how they should be built.

The development effort needed for a loosely-coupled system is a monotonically increasing function of the differences in the characteristics of the participating systems. Interfaces between logic programming systems and database systems are natural candidates for studying loose-coupling. This is because both types of systems are usually based on first order logic, and are thus similar in many respects. On the other hand, they are different enough so that developing a loosely-coupled interface between them is a challenging process.

Systems that incorporate the functionality of both logic programming and databases are called *Deductive Database Systems*. In addition to the loose-coupling approach, tight-coupling can be adopted in the development of such systems as well. Building from scratch, without any dependence on previously existing software, is also an option. The focus of this paper is loosely-coupled deductive database systems. Thus, the last two alternatives are

[1] We would like to thank Joanna Chen, Mark Friedman, Dennis Nara, and Mark Sturdevant for helping with the design and implementation of parts of BERMUDA at its various stages.

[2] This research has been partially supported by the National Science Foundation under Grant IRI-8703592.

[3] The second author has been partially supported by a NATO fellowship.

not discussed any further. The interested reader can find more information on them elsewhere [Gall78, Nais83, Nico83, Ston85, Morr86, Bocc86c, Naqv89, Ceri89b].

In this paper, we describe the design, implementation, and the results of an experimental performance evaluation of **BERMUDA** (**B**rains **E**mployed for **R**ules, **M**uscles **U**sed for **D**ata **A**ccess). BERMUDA is a loosely-coupled deductive database system that interfaces Prolog and the Britton-Lee Intelligent Database Machine 500 (IDM-500) [Ubel84] under the Unix operating system [Ritc78]. The inspiration for the name of BERMUDA was the envisioned programming environment where rules (and possibly small sets of facts) are stored in Prolog and (large sets of) facts are stored in a database. This, in turn, was motivated by the lack of any inferencing capability on the part of a database system as opposed to Prolog, and by the lack of any sophisticated secondary storage manipulation on the part of Prolog as opposed to a database system. The focus of this paper is on the particular architectural issues of building loosely-coupled systems and the resulting performance.

Several loosely-coupled interfaces between Prolog and relational database systems have been designed and/or developed [Jark84, Chan86, Ceri86, Deno86, Bocc86a, Bocc86b, Ceri89a]. A relatively detailed overview of some of them appears elsewhere [Ioan88]. Each one of the above systems suffers from one or more of the following shortcomings:

(a) The independence of the Prolog system has not been preserved — its interpreter has been changed.

(b) The use of the database system is not transparent to the user.

(c) Query answers are assumed to be small and are therefore stored in the virtual memory of Prolog.

(d) All database predicates of a Prolog clause are assumed to be grouped together.

(e) Only single relation queries are sent to the database — joins are processed by Prolog.

As it will become evident in the next sections, BERMUDA avoids all these problems. In addition, BERMUDA is not affected by any operating system limitations on the number of open files, pipes, or concurrent child processes, and it does not experience any performance degradation due to uncontrolled use of system resources. These were potential problems that were attributed to systems with the same approach as BERMUDA and their avoidance had motivated the design of some of the earlier systems [Bocc86b].

This paper is organized as follows. In Section 2, the design of BERMUDA is given. The key aspects of the system are discussed and the various design decisions are justified. This section is a slightly modified version of the corresponding section in the paper describing the first version of BERMUDA [Ioan88]. Section 3 describes the current implementation of BERMUDA. In Section 4, we discuss the performance of BERMUDA when a single query from a single Prolog process is executed. The performance of BERMUDA is compared against that of stand-alone Prolog and IDM. Also, the overhead imposed on the system by the modules of BERMUDA that implement the coupling between Prolog and IDM is discussed. Section 5 contains performance results of BERMUDA when multiple Prolog processes, each with multiple concurrent queries, are executed. In Section 6, we present a critique of the system, discuss the lessons that we have learned from the whole effort, and identify the aspects of the system that we would approach differently if we were to implement it again. Finally, Section 7 contains a summary of our work.

2. THE DESIGN OF BERMUDA

For the rest of the paper, a *database predicate* is a Prolog predicate that is stored as a relation in a database. Any other predicate is a *nondatabase predicate*. Equality (=), inequality ($\neq$) and all arithmetic comparison predicates ($>, <, \geq, \leq$) are *primitive predicates*. For a set of predicates P in a Prolog clause, the *associated primitive predicates* are those in the same clause that only involve variables that appear under the predicates in P. A pair of database predicates is *connected* if the two predicates share a common variable, or if one of them shares a common variable with a third predicate that is connected to the other one. A set of connected database predicates and possibly associated primitive predicates that appear consecutively in a Prolog clause interleaved with ANDs (,) and ORs (;) is a *database cluster*. A database cluster that does not have a database predicate or an associated primitive predicate immediately to its left or to its right is a *maximal database cluster*.

We refer to database queries that differ only on the constants that appear in them as queries that have the same *query template*. In IDM, the results of parsing and optimization of such queries are almost identical, differing only on the query constants. To expedite processing, if a certain option of IDM is used, given a query, IDM replaces its constants with some generic ones, parses and optimizes it in that form, and stores the outcome as a *stored command*. When a set of real constants is passed to a stored command, the latter is ready for execution without the need to repeat these two procedures. Stored commands are identified by *stored command ids*, which are returned when the former are first generated.

In all the forthcoming examples of Prolog programs, all database predicates are denoted by $\mathbf{d}_i$ and all nondatabase predicates are denoted by $\mathbf{p}_i$, where i is an integer. Also, for simplicity, we omit the arguments of the predicates whenever they play no role in the discussion. Note, however, that the arguments of database predicates are always assumed to be either constants or variables that can never be bound, directly or indirectly, to a list, since database systems do not support lists.

The current design of BERMUDA makes three assumptions:

- The order of access to tuples of database predicates is not important.
- Unless otherwise specified (by a cut (!)), users are interested in the complete set of answers to a query.
- There are no updates to database predicates, at least not during the execution of Prolog programs.

The first assumption is needed so that execution can be sped up by using different indices on database predicates for different queries. Presumably, database predicates are large, so the Prolog programmer does not know the order of the tuples and does not rely on that. The second assumption represents the canonical case in a database environment. The third assumption is motivated by simplicity. Combining the semantics of Prolog with the semantics of database updates generates several difficult problems. These are outside the scope of this work, since the focus of the development of BERMUDA has been on providing deductive capabilities over large databases and on studying the effectiveness of loosely-coupled systems at that.

Assuming the above, BERMUDA has been designed with the following goals in mind:

- It should truly preserve the independence of the end systems, i.e., both Prolog and IDM should remain unchanged.
- The existence of a database system underneath Prolog should be as transparent to the user as possible. This should hold both for how Prolog programs should be written and for how they behave when executed. In all cases, the impression given to the user should be that of interacting with a Prolog system.
- It should be as efficient as possible.
- It should allow sharing of data in the database among multiple Prolog programs.

The process structure of BERMUDA, as motivated by the above goals, is shown in Figure 2.1. Multiple Prolog processes communicate with one process called the *BERMUDA Agent* (BA). The BA in turn, communicates with a fixed number of identical *Loaders*. The Loaders contain two modules, the *Formater* and the *IDM Interface Library*. The former primarily translates queries and data between different representations. The latter is used for the communication with IDM, submitting queries to it and collecting the answers. For convenience, we

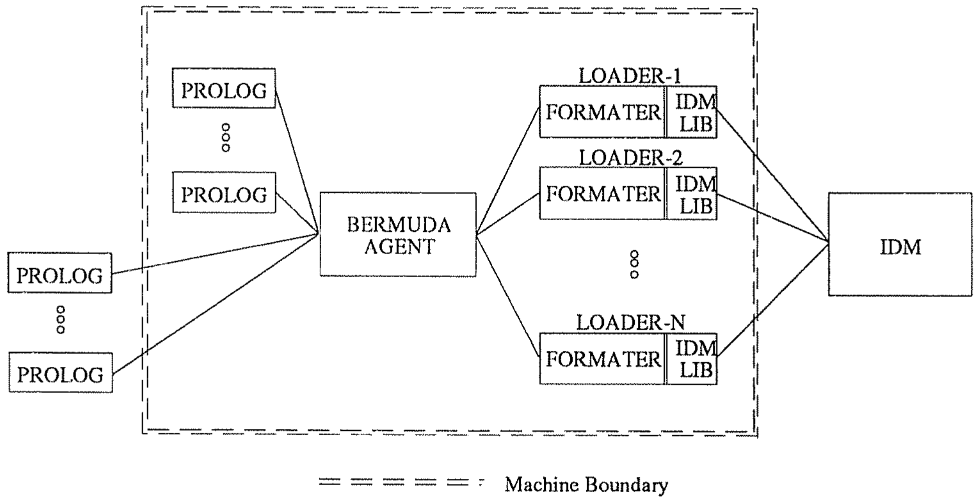

Figure 2.1: The process structure of BERMUDA.

occasionally refer to the BA and the Formater modules of the Loaders together as the *Connector Module*, since they comprise the software that we developed for connecting Prolog and IDM.

The flow of data in BERMUDA is the following. At run time, whenever Prolog encounters a maximal database cluster in a program clause, it sends some internal representation of the appropriate query to the BA. The BA sends the query to an idle Loader, if there is any, or puts it on a queue to wait until a Loader becomes free. The Loader formulates the appropriate SQL query, passes it on to IDM, and in due time it collects the answer, which is stored in a Unix file. The BA reads the file and starts providing Prolog with pages of tuples at the pace at which Prolog asks for them. There can be multiple queries being manipulated by the BA at the same time, coming either from the same Prolog process or from different ones.

The most significant aspects of the design of BERMUDA are analyzed in the following subsections. These are the following: separating rules from facts, sending to the database system as few queries as possible by using the maximal database cluster as the unit of transformation to a database query, caching query answers and stored command ids in the BA, prefetching query answers for Prolog, employing multiple Loaders, and handling the special, extralogical constructs of Prolog, like the cut (!).

2.1. Separating Rules from Data

The separation of rules and data is motivated by the goal of keeping Prolog and IDM unchanged. IDM cannot support any form of rules (with the exceptions of view definitions and integrity constraints, which are very simplified and restricted forms of rules supported by most relational database systems). Hence, any non-ground facts of a Prolog program have to remain under the control of Prolog and be stored within it. On the other hand, whenever a huge number of ground facts is associated with a predicate, it is wise to have the predicate stored as a relation under a database system. Thus, one can take advantage of the sophisticated manipulation of secondary storage offered by the database system and escape the generalities and performance deficiencies of the virtual memory of Prolog. Another advantage of storing large predicates in the database is that it makes these predicates persistent, without the need to load them every time a Prolog program is using them. Of course, it is left to the BERMUDA programmer to decide which predicates will be in the database and which ones will be in Prolog. We believe, however, that it is mostly the large predicates that both will tend to be persistent and will improve performance when stored in a database.

2.2. Sending Maximal Database Clusters as Queries to the Database System

In BERMUDA, a maximal database cluster is the unit of transformation of Prolog subclauses to queries to the database system. This is motivated by the following two advantages. First, like most relational database systems, IDM supports several query processing algorithms for joins, as opposed to Prolog, which supports a fixed one (essentially nested-loops [Seli79]), and it has the ability to use indices to expedite data access. In addition, it employs sophisticated optimization techniques to choose the most efficient algorithm for each query. Thus, in most cases, a query is executed much faster in IDM than in Prolog. Second, the overhead of sending a query through the Connector Module to IDM and then propagating the answer to Prolog in the opposite direction is not negligible. Sending a single query for each maximal database cluster minimizes the number of times this overhead is paid.

An alternative suggested by some other systems [Ceri86, Ceri89a] would be to send only single relation queries to IDM. This would have inferior join performance, however, since nested-loops would be the only join algorithm used. In addition, for every tuple of the outer relation a query would be issued to IDM to retrieve the matching tuples from the inner one. Every iteration of the outer loop would tend to be quite expensive because of the overhead of passing through the Connector Module and of the execution time of a full query by IDM. Therefore, assigning the join processing to the database system has important benefits for the overall performance.

Two examples of transforming a maximal database cluster into a single query are given below, one for a join and one for a selection. Assume that some Prolog program has to evaluate the following clause:

$$\mathbf{p}_1 :- \cdots , \mathbf{p}_2, \mathbf{d}_1, \mathbf{d}_2, \mathbf{p}_3, \cdots .$$

If "$\mathbf{d}_1,\mathbf{d}_2$" involves a join between $\mathbf{d}_1$ and $\mathbf{d}_2$, BERMUDA sends "$\mathbf{d}_1,\mathbf{d}_2$" as a single query to the database system, as opposed to sending multiple separate queries, one of the form "$\mathbf{d}_1$" and many of the form "$\mathbf{d}_2$" (as many as there are tuples in the answer of the first query). On the other hand, if "$\mathbf{d}_1,\mathbf{d}_2$" does not involve a join between $\mathbf{d}_1$ and $\mathbf{d}_2$ (i.e., it represents a cross product of the two predicates), BERMUDA sends two separate queries, "$\mathbf{d}_1$" and "$\mathbf{d}_2$". Similarly, assume that a Prolog program has to evaluate the following clause:

$$\mathbf{p}_1 :- \cdots , \mathbf{p}_2, \mathbf{d}_1(\ldots,X,\ldots), X>10, \mathbf{p}_3, \cdots .$$

BERMUDA will again choose to send "$\mathbf{d}_1(...,X,...),X>10$" as a single query to the database system. Any indices existing on the column of $\mathbf{d}_1$ where X appears can thus be used to apply the selection "$X>10$" and minimize processing time. Note that the same would happen if the selection appeared to the left of the database predicate: ordering of predicates is not important.

2.3. Caching Query Answers and Stored Command Ids

Caching of query answers by the BA is motivated by the need to avoid sending the same query to IDM multiple times. This can amount to very significant performance gains, because the backtracking mechanism of Prolog is a potential source of several such queries. For example, consider the following Prolog program:

$\mathbf{p}_1(X,Y) :- \mathbf{p}_2(X,Z),\mathbf{d}_1(Z,Y).$
$\mathbf{p}_2(2,1).$
$\mathbf{p}_2(3,1).$
$\mathbf{p}_2(4,5).$
$\mathbf{p}_1(X,Y)?$

According to the execution paradigm of Prolog, the query "$\mathbf{d}_1(1,Y)$" will be generated after $\mathbf{p}_2$ is resolved with its first ground clause, and the same query will be generated again after $\mathbf{p}_2$ is resolved with its second ground clause. It is much faster the second time if the answer to the query is simply retrieved from the cache instead of being reconstructed from scratch.

In BERMUDA, cached answers are kept in flat files that are accessible to the BA. Another alternative would be to cache query answers in the virtual memory of Prolog. Most likely, cache hits would be processed more efficiently in that case than in BERMUDA, but the cache would not be shared across Prolog processes. In addition, special purpose techniques could not be used for prefetching and buffering, and one would have to rely on the operating system for these services. These disadvantages were the reason for our decision to adopt external caching.

Caching of stored command ids is motivated by similar reasons as above. The backtracking mechanism of Prolog generates many queries with identical query templates. By having the corresponding stored command id cached in the BA, for all but the first such query, parsing and optimization in IDM can be avoided and performance improves. In the above example, the query that will be generated after $\mathbf{p}_2$ is resolved with its third ground clause is not a duplicate one, but it does have the same query template as the first two and can thus take advantage of a stored command.

2.4. Prefetching Query Answers

Prefetching of query answers by the BA is motivated by the need to correctly synchronize the processing of the various modules of BERMUDA. If no attention is paid to that, the differences in speed and processing model of these modules may degrade performance by forcing some of them to unnecessarily block while waiting for responses from others. This is especially crucial at the interface between Prolog and the BA, since Prolog accesses virtual memory, whereas the BA retrieves query answers from disk. The speed difference between virtual memory and disk may make the BA a potential performance bottleneck, forcing Prolog processes to block. Of course, this is a problem only with queries on which Prolog is slower than IDM. Otherwise, Prolog must wait anyway for IDM to finish processing. As an example, consider the following Prolog program:

$\mathbf{p}_1(X,Y) :- \mathbf{d}_1(X,Z),\mathbf{p}_2(Z,Y).$
$\mathbf{p}_2(2,1).$
$\mathbf{p}_2(3,2).$
...
$\mathbf{p}_2(10000,9999).$
$\mathbf{p}_1(X,Y)?$

Assume that, because $\mathbf{p}_2$ is large, Prolog is slower than IDM. Also assume that $\mathbf{d}_1$ is large enough so that the file containing the answer to the query "$\mathbf{d}_1(X,Y)$" spans several pages. After all the tuples of one page have been sent to and examined by Prolog, the next page has to be retrieved. Unless some provision is taken, Prolog will have to block waiting for an I/O to happen.

Prefetching in BERMUDA works as follows. As soon as the BA sends a page of a query answer to Prolog, it asks for the next one. If that page is already in the answer file, it is brought into the buffers of BA and is thus ready for the next request from Prolog. Thus, the I/O happens while Prolog manipulates the tuples of the previous page. When the first tuple in the new page is requested, it is already in the BA buffers, and Prolog does not have to wait.

2.5. Employing Multiple Loaders

Placing of Loader processes between the BA and IDM is motivated by the unavailability of a non-blocking interface to IDM and the lack of multiple threads in Unix processes. Whenever a query is sent to IDM, the sender has to block and wait for the first tuple in the answer to be prepared before running again and collecting it. Connecting the BA directly to IDM would thus require the BA to block every time a query was sent to the database system. This, in turn, would cause all the Prolog processes with database queries to wait until the BA came back again to accept the next query, effectively serializing their database accesses. BERMUDA avoids this problem, by having the Loaders block when a query is passed to IDM. The BA remains available for both accepting more queries from other Prolog processes and sending data back to them. When the Loader starts collecting the query answer, it notifies the BA, which in turn starts feeding the appropriate Prolog process with tuples from the answer. This creates a pipeline between Prolog and IDM, with Prolog manipulating the first tuples from the query answer before the complete answer is formed.

With respect to the Loader configuration, decisions have to be made on the number of the Loaders and on whether they should be created and destroyed on demand by the incoming database queries or they should live for the duration of the program. As justified below, our decision has been that there should be a fixed number of Loaders that live for the duration of the program, each one handling at most one query at a time.

Creating a new Loader process for every incoming query and destroying the process at the end is an expensive operation. This becomes significant if we take into account that a Prolog program, due to backtracking, may issue a huge number of database queries. Therefore, BERMUDA has a fixed number of Loaders that are created at system invocation time and remain alive until the session is ended. If at any point there are more queries sent to the BA than there are Loaders, then queries are put in a single priority queue in the BA and are served by the Loaders using a First-Come-First-Served policy. Our choice of this scheduling policy has been motivated by simplicity. We also expect that it is the optimal and fairest policy in most cases.

Clearly, with the above scheme the blocking problem of Prolog processes is not avoided completely, since queries can be placed in a queue. Thus, enough Loaders should exist to make waiting a rare event. On the other hand, too many Loaders can increase the resource contention with negative effects on performance. The optimum number of Loaders should keep a balance between the above two conflicting requirements. The performance results in Section 5 shed some light on what that number should be.

Note that even a single Prolog process can generate multiple simultaneously active database queries. This situation can be created by two types of clauses: recursive clauses and clauses with multiple maximal database clusters, each separated from the next by at least one nondatabase predicate. We give an example of the second type; recursive clauses behave similarly. Consider a Prolog process running a program containing the following clause:

$$\mathbf{p}_1 :- \cdots, \mathbf{p}_2, \mathbf{d}_1, \mathbf{p}_3, \mathbf{d}_2, \mathbf{p}_4, \cdots .$$

The Prolog process will first issue "$\mathbf{d}_1$" as a database query. As soon as it gets its first $\mathbf{d}_1$ tuple back, and after it evaluates $\mathbf{p}_3$, it will issue another query, "$\mathbf{d}_2$". This may happen while the answer to the first query "$\mathbf{d}_1$" is still being formed, thus leading to two simultaneously active queries issued by the same Prolog process.

2.6. Handling Extralogical Constructs of Prolog

In addition to "data" predicates, Prolog supports various extralogical predicates that are used to control the flow of data in a program. The most significant of those extralogical constructs of Prolog is the cut predicate (!). The appearance of ! in a Prolog program presents an important optimization challenge to BERMUDA, since queries with ! are not fully executed. The first instantiation of the variables that makes the partial clause before the ! evaluate to "true" is the only one needed. When this happens, part of the Prolog stack is thrown away, and the program never backtracks before the !. Hence, if there are database predicates before the !, it is highly probable that a large portion of the corresponding query answers will never be used. To minimize unnecessary extra work, the Prolog process sends a ! to the BA, which closes the corresponding scan on the query answer file. Ideally the BA should notify the appropriate Loader to stop collecting further tuples, if the complete answer has not already been formed, as they are not going to be used. The current version of BERMUDA, however, does not support this feature.

3. THE IMPLEMENTATION OF BERMUDA

BERMUDA has been implemented at the University of Wisconsin. The BA and the Loaders together with any Prolog application run on machines supporting Unix and communicate among themselves and with IDM over a 10Mb/sec local area network. Following a client/server model, communication between the BA (the server) and any other process (the client) is completely asynchronous — the BA never blocks to wait for another process. As we mentioned before, this does not apply to the communication between the Loaders and IDM, since no non-blocking interface to IDM is supported. In the following subsections, we discuss the implementation details of the various parts of the system.

3.1. Prolog

For the implementation of BERMUDA, we used C-Prolog 1.5+, since it was the only one available to us. This version allows for Prolog to call compiled C routines for execution, a capability that was very significant in the implementation of the system. Prolog itself cannot communicate with other processes, and this is a problem that has been faced by other similar projects as well [Bocc86b, Ghos88]. The capability of calling C routines, however, solves the problem by implementing all interprocess communication on the same machine or across different machines with such routines. Communication is achieved using Unix domain sockets for processes on the same machine and TCP sockets for processes on different machines.

With the exception of some necessary declarations in the beginning, Prolog programs written for BERMUDA do not explicitly refer to the underlying database system. BERMUDA takes care of the necessary database accesses. This is achieved by having every incoming Prolog program pass through a *Preprocessor*, before it is sent to the Prolog interpreter. This is shown in Figure 3.1.

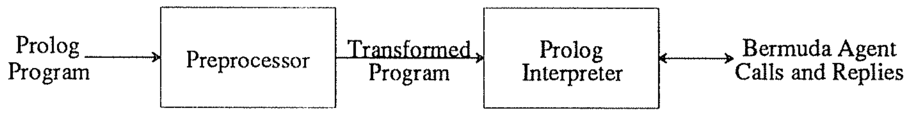

Figure 3.1: Preprocessing a Prolog program.

The preprocessor is implemented entirely in Prolog, and its sole purpose is to transform the original Prolog program at *consult*-ing[†] time by inserting the appropriate calls to the BA into it. The transformed program is then ready to be executed by the Prolog interpreter whenever an associated query is presented to the system.

The preprocessor follows the rule-based paradigm and works as follows. It makes several passes through the input program, always searching for patterns that conform to the antecedent of its internal set of rules. When one such pattern is found, then the relevant part of the program is rewritten according to the consequent of the corresponding rule. As long as the program has been rewritten during one pass over it, the preprocessor repeats the process; otherwise, it finishes. Essentially, every pass through the program identifies increasingly larger database clusters, until the largest one of them is found. The first pass also marks the database predicates that appear before any existing cut (!).

The preprocessor puts database queries in a canonical form. Specifically, given a maximal database cluster, the query that it represents is rewritten as follows. All database predicates are placed in the beginning in alphanumeric order, with their associated primitive predicates following at the end. Each constant is replaced by a placeholder of the form Ci, where i is some integer corresponding to the order in which the constant is found in the original database cluster. In addition, each variable is renamed into Vi, where i is an integer as above, with multiple instances of the same variable being mapped to the same new name. The product of the above transformations, structured as a Prolog list, is a *canonical query template*. The preprocessor replaces the original database cluster by the predicate **dbq** having as arguments the database to which the query must be sent, the above generated query template, a list of the original variable names in the order of the index i of their new names Vi, and the corresponding list of the original constants. Any bindings of the new variables in the query that are returned as answers from the database can be mapped back to the original variables by the appropriate variable list that has been passed to **dbq**. Processing of **dbq** goals at run time involves forming a *query tree* from the arguments of the corresponding instance of the **dbq** predicate and sending it to the BA.

For example, consider the following maximal database cluster in a Prolog clause, with predicates from a database called DB:

emp(john,S,D), S>100000, **dept**(D,7).

The corresponding canonical query template is constructed by rearranging the above predicates, renaming its variables and constants, and forming the following list:

Qtemp = [∧, [**dept**, V2, C3], [**emp**, C1, V1, V2], [>, V1, C2]].

[†] *consult* is a Prolog predicate that adds into a Prolog program a set of clauses written in a file.

The corresponding variable and constant lists are as follows:

Vars = [S, D],

Consts = [john, 100000, 7].

The above three lists together with the associated database name are the arguments of the **dbq** predicate that replaces the original database cluster in the Prolog program:

dbq(DB, Qtemp, Vars, Consts).

As discussed in Section 4, the above normalization process of the query form has minimal cost. Also, it represents a one-time penalty since it is done at preprocessing time. Moreover, it allows the run time cost to be significantly reduced by the simplification of the process of testing for equivalence of query templates or queries. The former simply requires a scan of the query templates from left to write looking for a character by character match. The latter requires the additional check for a match of the list of constants. This trade-off has benefited the overall performance of the system.

The task of the preprocessor is slightly more complicated in the presense of disjunctions. We decided to support only a limited form of them as parts of database clusters. Specifically, we allowed only disjunctions that can be expressed in a single, flat SQL statement with arbitrary nesting of ORs and ANDs in the qualification. Detecting such disjunctions is very easy. Every branch of the disjunction must be a conjunct of database predicates and an arbitrary boolean expression of associated primitive predicates. The database predicates must be the same in all branches, i.e., the database predicates must be the same and the joins between them must be the same.[†] The boolean expressions of associated primitive predicates can be arbitrarily different in the branches of the disjunction. The equivalent SQL query of any Prolog disjunction that satisfies the above is easily constructed. The target list contains all attributes of the (common) database predicates of the branches, which predicates appear in the from-clause of the statement. The qualification contains all appropriate joins among the database predicates and the disjunction of the boolean expressions of primitive predicates of the branches.

As an example, consider the following Prolog disjunction, which satisfies all the above mentioned restrictions:

$$\mathbf{d}_1(X,Y), ((\mathbf{d}_2(Y,Z), X=10, Z<100); (\mathbf{d}_2(Y,Z), X=20, (Z=30; Z=40))).$$

For simplicity assume that the Prolog variables above are the same with the names of the corresponding attributes in the database. Then, the equivalent SQL query is

select $X, \mathbf{d}_1.Y, Z$

from $\mathbf{d}_1, \mathbf{d}_2$

where $\mathbf{d}_1.Y=\mathbf{d}_2.Y$ and ((X=10 and Z<100) or (X=20 and (Z=30 or Z=40))).

3.2. The Bermuda Agent

The BA primarily serves the role of "postman" between IDM and Prolog. It achieves that by using the data structures that are shown in Figure 3.2. *Open Scans* contains information about each active query and the originating Prolog process. *Busy Loaders* contains information about which Loaders are busy and with which query. *Past Queries and Query Templates* contains information about all queries that have been sent to IDM in the past and their query templates. It is a two level data structure with the primary level being a hash table on query templates. Associated with each query template there is a secondary level hash table on lists of constants that correspond to the placeholders of the form Ci in the template. If a query matches an entry of the structure in only the first level, then a stored command can be used to process it. If a query matches an entry of the structure in both levels, then the cache can be used to retrieve its answer directly. Old query answers are removed from the cache based on an LRU algorithm.

When a Prolog process sends a query tree to the BA, the latter makes an entry into *Open Scans*. The BA then looks into *Past Queries and Query Templates* to see if the corresponding query or one with the same template has been answered before. In the first case, the appropriate file is opened, and tuples are retrieved to be passed to Prolog (unless the first page of the file is already in the buffer pool, in which case tuples may be immediately returned to Prolog). In the second case, the query tree is sent to an idle Loader together with a stored command identifier, so that it avoids being parsed and optimized again. If neither the query nor the query template have been seen before, the query tree alone is sent to an idle Loader. Except when the cache is used, if there is no idle Loader, the query

[†] Actually, even if the joins are different, the construction of the equivalent flat SQL query is not hard. Allowing such disjunctions, however, would add some further complications to the preprocessor in handling cases where one branch has a cross product instead of a join (Section 2.2). Thus, we decided to exclude this case completely.

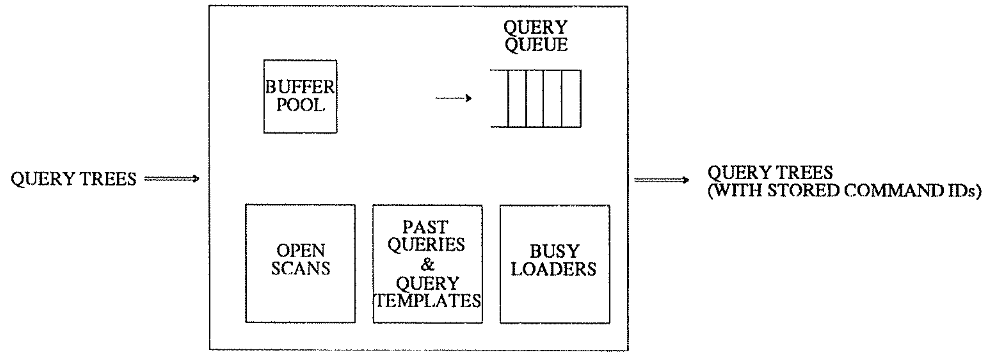
QUERY QUEUE
BUFFER POOL
QUERY TREES
QUERY TREES (WITH STORED COMMAND IDs)
OPEN SCANS
PAST QUERIES & QUERY TEMPLATES
BUSY LOADERS

Figure 3.2: Data structures of the BA.

tree is put at the end of the query queue to wait for one to become available. Also, if the query tree appears in the queue already or is currently being processed by a Loader, a note of this fact is made so that it is not sent to IDM again.

When the Loader has collected a page's worth of the answer, it stores it into a Unix file and then notifies the BA, which in turn passes it to Prolog through a pipe. After that, whenever Prolog asks for a new page, the BA passes it on. While this is happening, the Loader may still be appending more of the query answer to the Unix file. When the complete answer has been written out, the Loader notifies the BA that it has become available and is removed from *Busy Loaders*.

The BA maintains a buffer pool with pages of various query answers that are requested from Prolog processes. Each scan from *Open Scans* is assigned one buffer. Several scans may point to the same buffer, although sharing is expected to be rare. As described earlier, the BA prefetches pages from a query answer so that Prolog does not have to wait for I/O. Whenever a page is sent to Prolog, it is replaced by the next one in the answer file whenever that becomes available, or immediately if it has been produced already. If the number of scans becomes grater than the number of available buffers, then the page of the least recently used scan is replaced by the first page of the new one.

3.3. The Loaders

As we have described earlier, the purpose of the Loaders is to cope with the lack of a non-blocking interface to IDM, and to take advantage of the capabilities of IDM to handle multiple concurrent users. Each Loader consists of the Formater, which has been implemented by us, and a library provided by IDM that is used for the communication with the back-end of the system. An idle Loader receives either a query tree or a query tree and a stored command identifier from the BA. In the first case, if the query contains relations that are accessed for the first time, the Formater constructs an auxiliary query and sends it to IDM to retrieve the corresponding attribute names. Otherwise, this information is kept in an attribute cache in the Formater, and the auxiliary query is avoided. The query tree is then transformed into an SQL statement [Astr76], which in turn, is used to create a new stored command. In the second case, all the above steps are bypassed, because a stored command exists already in IDM. In both cases, after a stored command is generated, the actual constants from the query tree are used so that an executable query is formed. The Loader sends this query to IDM and then blocks. When IDM is ready with the first tuple in the answer, it notifies the Loader, which wakes up and starts collecting it. The IDM library allows only one-tuple-at-a-time retrieval of the answer. To speed up servicing the other parts of the system, the Formater continuously asks for tuples until it fills a page. It then writes the page in a specified Unix file, notifies the BA that the answer is partially ready, and turns back to IDM to continue collecting the query answer. When the complete query has been written into the file, it notifies the BA and receives the next query for processing (if there is one waiting in the queue).

4. PERFORMANCE EVALUATION OF SINGLE QUERIES

In this section, we present the results of a series of experiments that we have performed when individual queries are submitted to BERMUDA. All experiments have been performed on a microVax 3200 with 8 Mbytes of main memory under Unix 4.3 BSD, which communicated with an IDM-500 with 1 Mbyte of main memory. The block size that we have used has been 4 Kbytes. The experiments can be divided into two categories, benchmarks and profiles. That is, in the first category of experiments, we compare the performance of BERMUDA with that of stand-alone Prolog and stand-alone IDM. In the second category of experiments, we compare the cost of the individual modules of BERMUDA, i.e., the Prolog process, the Connector Module, and the IDM. The results of these categories of experiments are described in the following two subsections.

4.1. Benchmarks

The Wisconsin Benchmark [Bitt83] is widely accepted as a good set of queries to test the performance of database systems. Hence, unlike other projects that have developed specialized benchmarks [Will87], we have adopted the Wisconsin Benchmark for comparing the performance of BERMUDA, Prolog, and IDM. We have concentrated on selection and join queries, because they are the ones that can be run on all three systems and they are the most common. In all these experiments, IDM has run the original SQL query of the benchmark, whereas Prolog and BERMUDA have run the equivalent Prolog query. Details on the Wisconsin benchmark can be found elsewhere [Bitt83]. The following are the most important parameters of the used relations.

Parameter	Value
Relation Size (tuples)	1000 or 10000
Tuple size (bytes)	182
Value distribution	uniform

Table 4.1: Database parameters.

All relations have the same schema, i.e., the same number and the same types of attributes. The queries that have been run are shown in Table 4.2. Queries Q0 to Q4 are selection queries (we include the null queries in those) and queries Q5 to Q9 are join queries, some of which involve selections as well. All the selected and joined attributes are either integers or character strings. The result of each query contains all attributes of all relations that participate in the query.

Query name	Query type
Q0	Null query.
Q1	1 tuple integer selection from 10K-tuple relation.
Q2	100 tuple integer selection (1%) from 10K-tuple relation.
Q3	100 tuple string selection (1%) from 10K-tuple relation.
Q4	1000 tuple integer selection (10%) from 10K-tuple relation.
Q5	Integer join of two 1K-tuple relations on primary keys.
Q6	Integer join of two 10K-tuple and one 1K-tuple relations on primary keys with 10% selection on two of the keys.
Q7	Integer join of one 10K-tuple and one 1K-tuple relations on primary keys.
Q8	Integer join of two 10K-tuple relations on primary keys with 10% selection on one of the keys.
Q9	String join of two 10K-tuple relations on primary keys with 10% integer selection on one primary key.

Table 4.2: Benchmark queries.

To ensure the validity of the results, for each system, selection queries were repeated 80 times, whereas join queries were repeated 16 times. All the numbers shown below represent the corresponding averages. The variance among these runs has been very small. Also, all times shown below are in seconds. Finally, in this subsection only, we use the term B-Prolog when referring to the Prolog process of BERMUDA, so that it is distinguished from stand-alone Prolog. Similarly, we use the term B-IDM when referring to IDM within BERMUDA.

Before we compare the execution times of the three systems on the above queries we want to discuss some execution overhead that Prolog and BERMUDA incur. In particular, for Prolog, this is the time spent in *consult*-ing, i.e., in loading the necessary relations in the address space of Prolog. Similarly, for BERMUDA, this is the time spent by the preprocessor. Table 4.3 shows the corresponding times for each query above. It is evident that Prolog *consult*-ing time is a nontrivial overhead. It grows at a rate that is slightly higher than linear, because the more tuples that are loaded into Prolog, the more time is spent in searching its internal memory (hashing on predicate name). It roughly corresponds, however, to spending 50 msec per loaded tuple. As it will become evident in the following paragraphs, this is rather significant compared to execution cost, for selection queries and small join queries. BERMUDA preprocessing time is also nonnegligible. As expected from the discussion in Section 3.1, it grows with the number of predicates in the query, since the latter determines the number of passes that the

Query	Prolog *consult*-ing (sec)	BERMUDA preprocessing (sec)
Q0	0.0	1.50
Q1	268.40	1.66
Q2	268.40	1.85
Q3	268.40	1.79
Q4	268.40	1.83
Q5	48.33	3.82
Q6	565.72	7.82
Q7	294.82	3.83
Q8	536.81	4.15
Q9	536.81	3.95

Table 4.3: Overhead of Prolog *consult*-ing and BERMUDA preprocessing.

preprocessor has to make over the query. Nevertheless, it will become evident in the following paragraphs that preprocessing represents a very small fraction of the execution cost of BERMUDA and can be ignored. For uniformity, all subsequent results for Prolog and BERMUDA do not include *consult*-ing and preprocessing times. They represent execution cost alone.

Figure 4.1 shows the elapsed time (y-axis) of each query above (x-axis) when executed in Prolog, BERMUDA, and IDM. As in all subsequent cases, for ease of presentation, separate graphs are given for selections and joins. The results shown are when no special feature of BERMUDA or IDM are used to improve performance, i.e., no indices, stored commands, or cache are used.

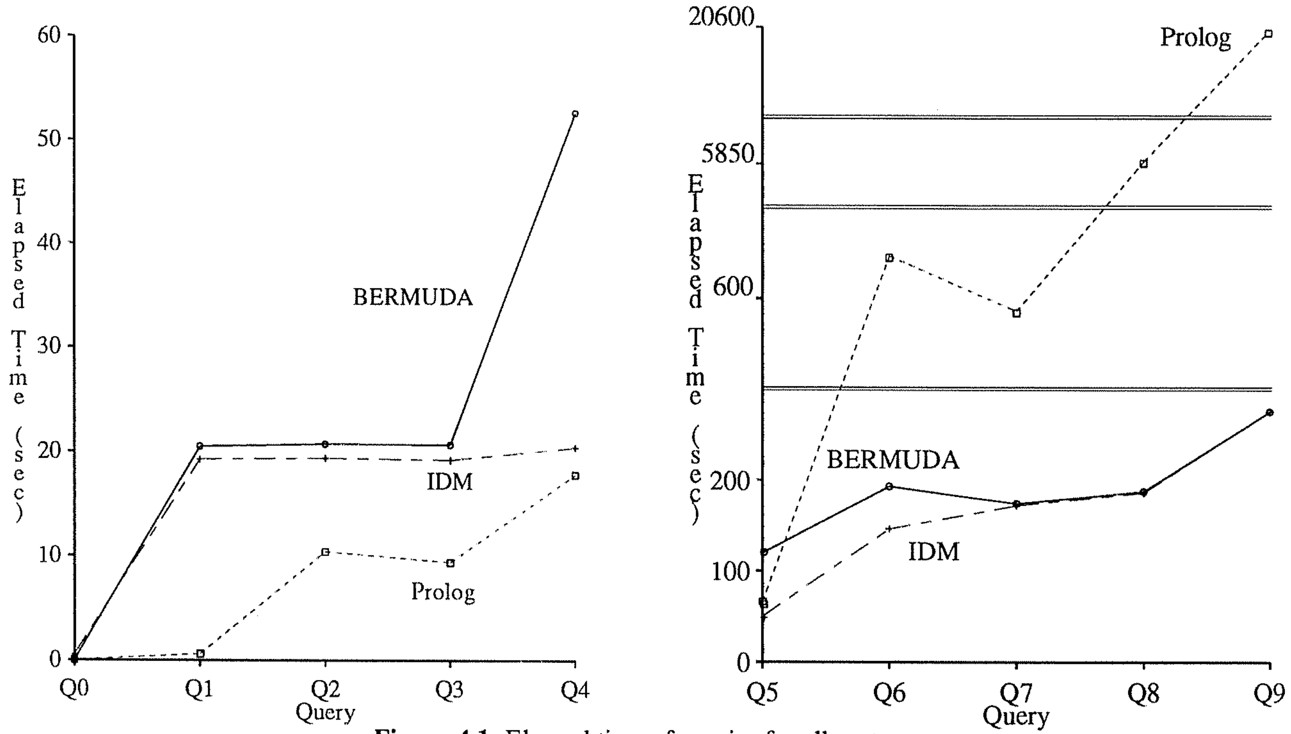

Figure 4.1: Elapsed time of queries for all systems when no indices, stored commands, or cache are used.

The most important observations from Figure 4.1 are the following. For selections, Prolog takes advantage of having everything into virtual memory and is superior in performance. On the other hand, for joins, except when very small relations are involved (Q5), Prolog has very bad performance, and the benefits of using database technology are tremendous.

The comparison of BERMUDA and IDM brings up some very interesting points. The performance of the former depends on two factors. First, it depends on the relative costs of B-Prolog and B-IDM at steady state, i.e., after the first page of results has been produced by IDM. Second, when B-Prolog is slower than B-IDM, it depends on the answer size as well. More specifically, when B-Prolog is faster than B-IDM, as expected, BERMUDA and IDM have similar performance. This is because B-Prolog is waiting for IDM all the time, and when the latter produces the last page of the answer, all that remains for BERMUDA to do is processing of that page by B-Prolog. When B-Prolog is slower than B-IDM, the largest the result size is, the worse BERMUDA becomes compared to IDM. This is because IDM finishes early, and beyond that point, processing by B-Prolog is the only activity in BERMUDA. The more pages that need to be processed during that time, the higher the difference in cost between IDM and BERMUDA.

All the above are very clear in Figure 4.1. For all selections (Q1-Q4), IDM has almost identical performance — it just scans a 10K-tuple relation. Moreover, B-Prolog is slower than B-IDM. Thus, BERMUDA has competitive performance when the result size is small (Q1-Q3), but it behaves poorly when the result size becomes significant (Q4). For all joins (Q5-Q9), the result contains 1000 tuples, but in terms of pages, the result of Q6 is larger than all others because its schema consists of the attributes of three relations instead of two. For the first two joins, B-Prolog is slower than B-IDM, and therefore, BERMUDA is losing in performance. As the cost of B-IDM (and IDM) increases, however, BERMUDA becomes more competitive, and for the three most expensive queries, it exhibits essentially equivalent performance to IDM. We should emphasize that the relative speed of B-Prolog and B-IDM is meaningful only for the steady state of the system. Query Q6 is rather expensive and one would expect for BERMUDA to have a more similar performance to IDM than what is observed. A large part of its cost, however, is spent before the first tuples of the answer are produced. Because of the presence of selections, at steady state, the query behaves like a join between 1K-tuple relations. This is why the difference between BERMUDA and IDM on Q6 is closer to their difference on Q5 than to their difference on Q7.

Figure 4.2 shows the effect of using indices (clustered and nonclustered B $^+$-trees). For each BERMUDA and IDM query, elapsed times when using no index, clustered indices, and nonclustered indices are given. When queries were executed with indices, depending on the query, indices existed on the following attributes. For all selection queries (Q0-Q4), an index existed on the selected attribute. For all integer join queries that also contained selections (Q6, Q8), an index existed on the join attribute of each 10K-tuple relation (all selected attributes are included in those). For all integer join queries that did not contain selections (Q5, Q7), an index existed on the join attribute of the 10K-tuple relation (if any), or on the join attribute of one of the 1K-tuple relations. For the string join query (Q9), an index existed on each join attribute. We should emphasize, of course, that the existence of these indices does not imply that they are used also, the decision lying with the query optimizer. For all queries tested, however, Figure 4.2 shows that indices were always used when available.

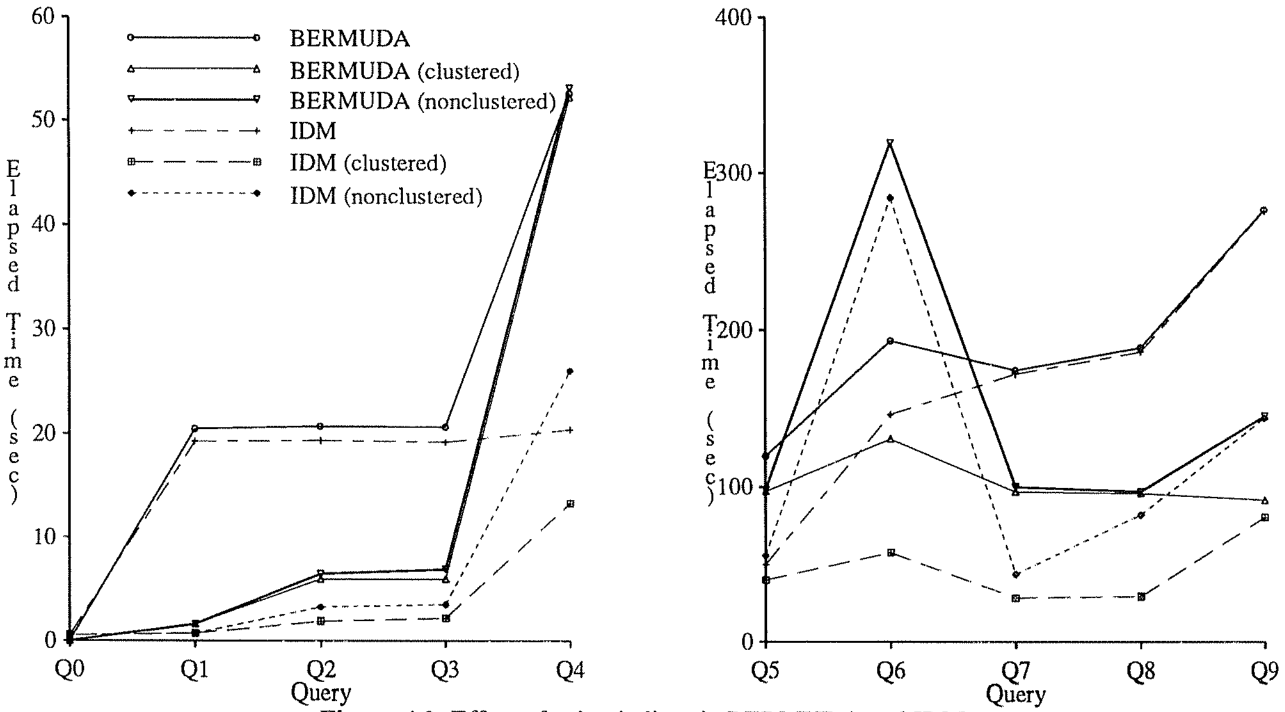

Figure 4.2: Effect of using indices in BERMUDA and IDM when no stored commands or cache is used.

The dependence of the performance of BERMUDA on the two factors that we mentioned when explaining the results of Figure 4.1 is evident in Figure 4.2 as well and is discussed no further. One interesting observation is that when indices are available, quite often BERMUDA and IDM become faster than Prolog even on selections (Q2, Q3). Also note that sometimes the query optimizer chooses the wrong plan. This is seen in queries Q4, Q5, and Q6, for the case where nonclustered indices are available. The results indicate that these indices are indeed used (the cost is different from the no index case), although using no index has better performance.

Table 4.4 shows the effect of using stored commands on the elapsed time of BERMUDA. For all queries, we show averages for the additional cost to generate a stored command and for the savings when using an existing stored command, compared to the regular cost of the queries. The results shown are for the case when the appropriate indices exist. The no index case has produced similar results.

Query	Overhead of generating stored command (sec)	Gain of using stored command (sec)
Q1	0.30	0.44
Q2	0.33	0.42
Q3	0.31	0.42
Q4	0.30	0.42
Q5	0.19	0.88
Q6	0.51	1.40
Q7	0.29	0.81
Q8	0.40	0.81
Q9	0.25	0.80

Table 4.4: Effect of using stored commands in BERMUDA.

The most important observations from Table 4.4 are the following. The overhead to generate a stored command is smaller than the savings of using one. Thus, if a query template is expected to be used more than once, it is beneficial to slow down the first query with that template, so that all subsequent ones are faster. We should also mention, however, that the gain of using a stored command is very small compared to the execution cost of queries. This implies that a query template must be repeated several times before the use of a stored command can affect performance significantly. In general, as verified by the results in Table 4.4, the overhead and the gain depend on the complexity of the query, i.e., the number and type of selections and joins, and the complexity of the physical design of the database, i.e., the number and type of available indices.

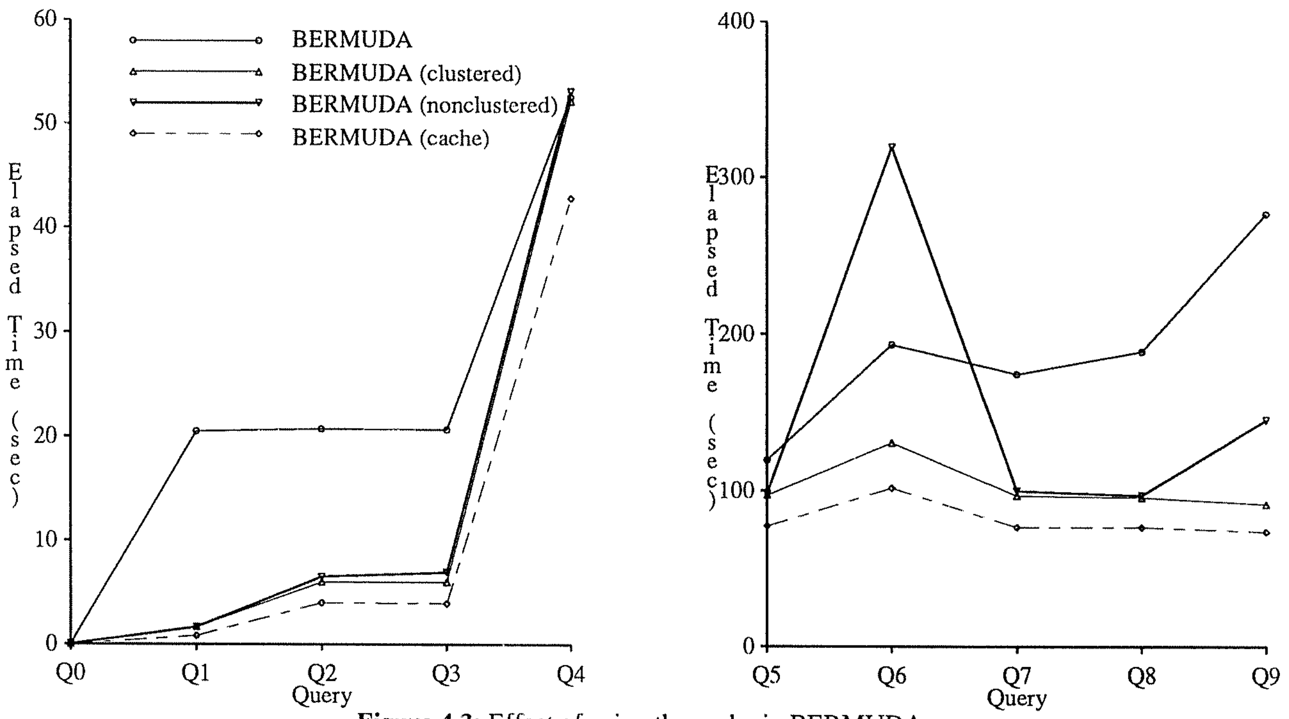

Figure 4.3: Effect of using the cache in BERMUDA.

Finally, we also show the effect of using a cached result on the performance of BERMUDA. Figure 4.3 contains the elapsed time of BERMUDA (y-axis) of each query (x-axis) when executed without a cached result for all three possibilities of index availability and when executed with a cached result. As expected, using the cache is always beneficial. The higher the query execution cost, the more the benefit of using the cache. One can also clearly see the effect of the result size on the cost of using the cache. For example, Q4 has ten times the result size of Q2 or Q3, and its cost is analogously higher. (In fact, the cost increase is more than linear due to constant overheads.) Similarly, the tuples in the result of Q6 have attributes from three relations as opposed to those in the results of Q5 or Q7-Q9 that involve two relations only. Since all results have 1000 tuples, the number of pages with which Q6 deals is larger, and therefore, its cost is higher.

4.2. Profiles

An important question on the performance of BERMUDA that cannot be answered from the benchmark results is how much of its time is spent in the Connector Module. Measuring that overhead provides a good indication of whether the Connector Module can ever be the bottleneck or not. This section presents results on the cost of distinct processes within BERMUDA that are helpful in estimating that overhead. For convenience, in this subsection, the terms ''Prolog'' and ''IDM'' refer to the corresponding modules of BERMUDA and not to stand-alone systems as in Section 4.1.

For the experiments of this subsection, we have used a ''fake'' Prolog process as the query provider and tuple consumer on the front end of the system. This process simulates Prolog accurately on all accounts except for its memory requirements, since when tuples are returned to it, they are not placed in any internal structures. This only affects the time consumed for memory searching, which remains zero in ''fake'' Prolog, whereas it increases as more tuples are inserted in actual Prolog. Experiments that we have conducted with ''fake'' and actual Prolog have shown no effect on the overall performance of BERMUDA and on the quality of the results presented below. Hence, we have used the ''fake'' Prolog because of the added flexibility in the experimentation process. The most important benefit of doing so is the freedom to specify arbitrary amounts of time for Prolog to spend on each incoming tuple of the answer. Thus, we have easily moved from systems where Prolog is the bottleneck to ones where IDM is the bottleneck and have observed the variations that were effected on the behavior of BERMUDA. Another benefit of using a ''fake'' Prolog process is that our findings are not tied to Prolog specifically but provide insights for loosely-coupled systems in general, thus satisfying one of the original goals of this study.

The time consumed by the Connector Module can only be approximately measured. This is due to several reasons. The presense of multiple processes makes CPU time an inadequate measure of the cost of individual processes. On the other hand, the operating system can start processes (deamons) that consume resources and increase what is measured as the elapsed time of processes unpredictably. Also, accounting for communication cost is hard, since it is measured in no process. Moreover, the services that Unix offers for time measurements are inadequate. The clock accuracy on the VAX processor is rather coarse and obtaining time measurements requires system calls that are costly since they imply context switches. Also, the measured overhead of the Connector Module happens to be close to the quantum of the operating system's scheduler, so stopping a process for scheduling reasons means a relatively large increase of the elapsed time of the process. For the above reasons, the results presented in this subsection should be accepted as estimates and not as accurate measurements. We do present them, however, because they provide a good indication of the order of magnitude of the cost.

4.2.1. Time Behavior

We have instrumented the various processes of BERMUDA so that, for a variety of event types, the time at which each one occurs is recorded. This permits monitoring the overall activity in the system and observing its behavior. Specifically, we keep track of the following events:

(1) Prolog receiving a page with tuples from the answer and starting processing them.

(2) Prolog finishing processing a page and requesting the next one.

(3) The BA receiving a message from Prolog requesting some page of the answer.

(4) The BA writing a page of the answer to the pipe that implements the connection to Prolog.

(5) The BA requesting some page from the answer file to be prefetched into memory.

(6) A prefetched page actually being transferred to memory.

(7) IDM starting the generation of tuples in some page of the answer.

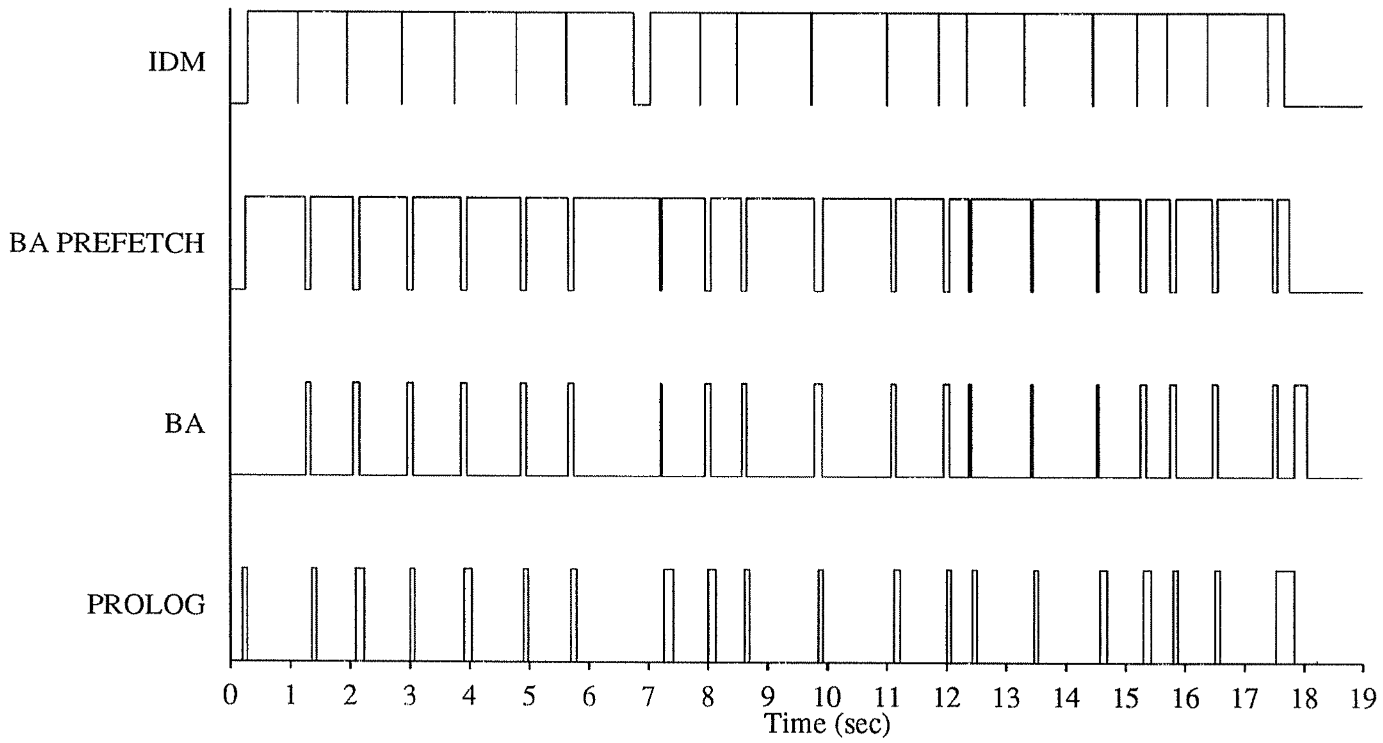

Figure 4.4: Activity in the modules of BERMUDA when Prolog is faster than IDM.

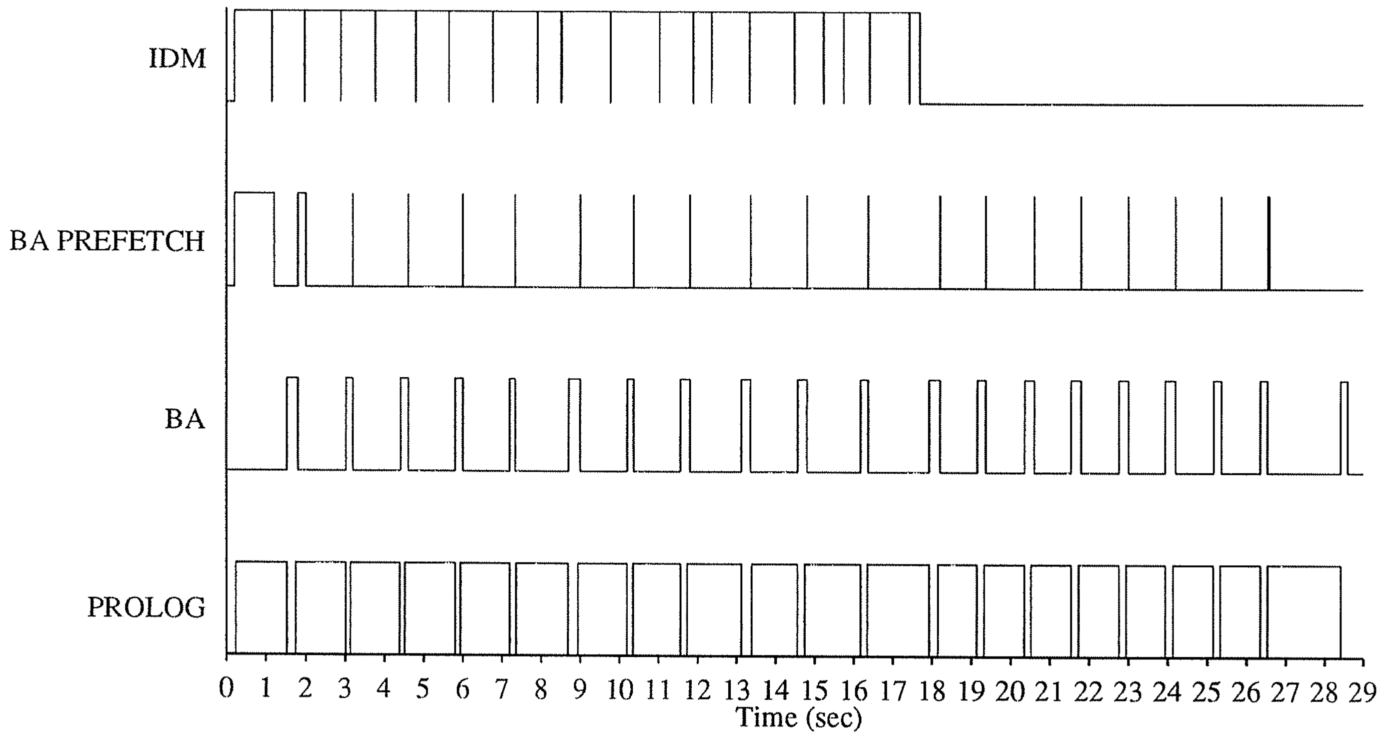

Figure 4.5: Activity in the modules of BERMUDA when Prolog is slower than IDM.

(8) IDM sending the last tuple of a page in the answer to the Loader.

From the interleaved occurrence of events of the above types, we are able to monitor when the various parts of the system are active and when they are waiting for some action in the rest of the system. Events of type (1) and (2) monitor the activity of Prolog, events of type (3) and (4) do that for the BA, and events of type (7) and (8) do that for IDM. Also, events of type (5) and (6) monitor the prefetching part of the BA.

We present two examples of monitoring the behavior of these parts of the system using the above event types. In the first example, Prolog needs 2 msec to process each tuple in the answer, and that makes it faster than IDM. In the second example, Prolog needs 100 msec for that and is slower than IDM. In both examples, the query executed is a selection query on a 10K-tuple relation without an index returning 21 pages. The time series of events that correspond to these examples are shown in Figures 4.4 and 4.5. The x-axis of these figures represents real time in seconds. The period during which IDM is working to produce the tuples of the first page in the answer is not shown, because all other parts of the system are waiting for it. We only show the activity after Prolog receives the first page, which in some sense, is the beginning of the steady state for the system. In Figures 4.4 and 4.5, for each one of Prolog, the BA, and IDM, a high value indicates activity with some page in the answer, whereas a low value indicates waiting. We should emphasize that a high value at some point in time does not mean that the associated process has the CPU and is running. It only means that, *whenever* the associated process controls the CPU during that period, it deals with a particular page. Finally, for the prefetching part of the BA, a high value indicates that a page has been requested from the file which is not there yet, whereas a low value indicates that a page has already been prefetched but it has not been requested by Prolog yet.

The following observations are in order. When Prolog is fast, Prolog and the BA wait for IDM. This can be seen by the low utilization of the Prolog process and by the long periods at which BA prefetching is in effect. When Prolog is slow, IDM finishes quickly (it is idle beyond a certain point), and overall progress is determined by the speed of Prolog. Prefetching is almost instantaneous because pages are always available, and the utilization of the Prolog process is high. Thus, the system behaves as expected. We should also like to comment on the period between the 6th and 7th second in Figure 4.4 when no useful work appears to be done: all three of Prolog, the BA, and IDM are idle, and prefetching does not occur. This is due to some system process stealing the CPU from BERMUDA and delaying all processes. The effect of such disturbances on all results presented in this paper has been minimized by performing experiments multiple times and ensuring that the variance in the output is low. Such events, however, cannot be completely prevented from happening.

4.2.2. Overhead of the Connector Module

The overhead of the Connector Module has been calculated using the timing of some of the events discussed in Section 4.2.1. We only present results for the case where Prolog is faster than IDM. This is because much fewer parameters are needed to compute the overhead in this case than when Prolog is slower than IDM, and therefore, the error introduced in the computation is lower. Also, we expect that the actual overhead is not significantly different in the two cases. Three parameters have been used in our computations.

T_I^n	The instance of time that IDM makes the last tuple of the n-th page available (event type (8)).
T_P^{n-1}	The instance of time that Prolog finishes processing the $(n-1)$-th page (event type (2)).
T_R^n	The instance of time that Prolog starts processing the n-th page (event type (1)).

Figure 4.6 presents a set of actions on a single page similar to those in Figure 4.4 indicating the above parameters.

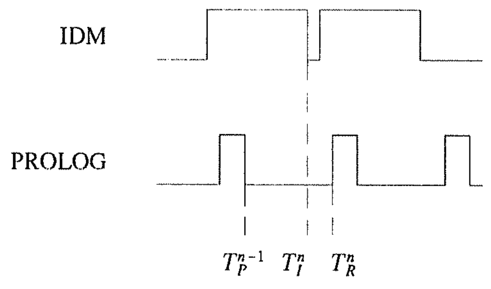

Figure 4.6: Timing parameters for the Connector Module when Prolog is faster than IDM.

The decision on the relative speed of Prolog has been made separately for each page as follows. If $T_I^n > T_P^{n-1}$ Prolog is faster than IDM for that page. If $T_I^n < T_P^{n-1}$ Prolog is slower than IDM. Let OV be the time consumed by the Connector Module on a page. From Figure 4.6, the formula for OV when Prolog is faster than IDM is given by

$$OV = T_R^n - T_I^n.$$

This is straightforward, since from the time that IDM prepares a page until the time that Prolog starts executing, the BA and the Loader are essentially the only two processes operating. Moreover, the Loader executes code of the Formater only. Clearly, the above formula calculates an upper bound on the actual time consumed in the Connector Module, because between T_I^n and T_R^n some time is often consumed inside the IDM library for starting up processing of the next page. This is accounted against the cost of the Connector Module in the above formula. Moreover, the cost of context switches, any communication overhead, and also the time consumed by any system process that takes control of the CPU are accounted against the Connector Module. Thus, the numbers shown below represent estimates of the worst case for the measured overhead and are slightly pessimistic.

We have experimented with BERMUDA using ''fake'' Prolog and obtained measurements for the parameters defined above. The results presented below are for a selection query on a 10K-tuple relation without an index. The number of pages returned by the query has been varied, but this has not affected the system behavior. The processing time of Prolog has been varied also for the same reasons. All results presented below are averages of a large number of measurements. The variance among similar measurements has been small.

Prolog cost (msec/tuple)	Prolog cost (msec/page)	Connector Module overhead (msec/page)
2	80	114
10	240	118
50	1000	125
100	1930	112

Table 4.5: Overhead of Connector Module
as a function of the processing cost of Prolog.

Table 4.5 shows the overhead of the Connector Module for several values of the processing cost of Prolog. The latter has been controlled by varying the CPU cost consumed by the Prolog process per tuple, but we also show the measured CPU time consumed per page, which is more directly comparable to the overhead measurements. All times are given in msec. Clearly, the computed upper bound on the overhead of the Connector Module is rather small. Moreover, as expected, it remains relatively unaffected by the processing cost of Prolog, and is equal to about 120 msec per page.

5. PERFORMANCE EVALUATION OF MULTIPLE QUERIES

All experiments described in the previous section deal with a single Prolog process submitting a single query to the database. As a consequence, a single Loader has been enough to handle this case. In this section, we present the results of some experiments that explore some additional parameters of the system. Specifically, we study the performance of the system when the values of the following parameters are varied: the number of maximal database clusters in the same Prolog clause (denoted by C), the number of concurrent Prolog processes (denoted by PP), and the number of available Loaders (denoted by L). All times in this section are given in seconds.

In these experiments, we have used again the ''fake'' Prolog for the reasons that were discussed in Section 4. In order for our measurements to be comparable, each Prolog process evaluates a single clause. Also, all queries, whether from the same Prolog process or different ones, are equivalent. Specifically, they are selections on a 1K-tuple relation without an index. Each query returns 6 pages, and every Prolog process consumes approximately 430 msec for every page in the answer (20 msec per tuple). Experiments with several other values for the cost of Prolog have given similar results.

In designing this experiment, special attention had to be given on relating C with the overall load generated by each Prolog process to the system. Specifically, consider a Prolog clause with C maximal database clusters. Let S_Q be the number of queries generated from that clause that are simultaneously active in the system at some point in time, and let T_Q be the total number of queries generated from that clause. Under the assumption that the answer to the query represented by the entire clause is nonempty, these two numbers are related to C with the following inequalities:

$$S_Q \le C \le T_Q. \qquad \textbf{(5.1)}$$

Because of our desire to control the generated load by specifying only C among the three parameters of (5.1) (in addition to PP and L), we have set up the experiment so that it operates at the point where both parts of the above formula hold with equality. Specifically, to satisfy $C{=}T_Q$, we have assumed that, from each query answer, only a single tuple provides successful bindings to intermediate nondatabase predicates to cause the query that corresponds to the next database cluster to be executed. Note that, by definition, there is at least one nondatabase predicate between two consecutive maximal database clusters. To satisfy $S_Q{=}C$, the queries of all database clusters in the Prolog clause must be simultaneously active. Clearly, this cannot hold for the entire period of processing of the clause. To maximize the duration of activity under full load, queries have been artificially started almost immediately one after the other. This guaranties that all of them fight for Loaders simultaneously. The tuples in their answers, however, are consumed by the Prolog process in the proper order according to the canonical processing paradigm of Prolog. This implies that, as queries finish, the equality of S_Q and C is lost ($S_Q < C$), system load decreases, and the processing of the remaining queries are expedited. This is unavoidable, however, due to the Prolog processing scheme.

Figure 5.1 sheds some light on the potential parallelism that exists in the system between processing in Prolog and IDM. Specifically, it shows the elapsed time of Prolog processes as a function of T_Q, the total number of generated queries. To remain unaffected from parallelism among queries within IDM, all experiments have been run with a single Loader. In each case, the desired number of queries T_Q has been generated in two different ways, which correspond to the two extremes: using a single Prolog process with T_Q maximal database clusters in its clause ($PP{=}1$ and $C{=}T_Q$); using T_Q Prolog processes, each one evaluating a clause with a single maximal database cluster in it ($PP{=}T_Q$ and $C{=}1$). The performance of intermediate cases has fallen between these two extremes. For the case of multiple Prolog processes, Figure 5.1 shows the range of elapsed times of the processes, as well as the average such time. Also, the dotted line indicates the total execution time of queries if they are executed sequentially, with no parallelism.

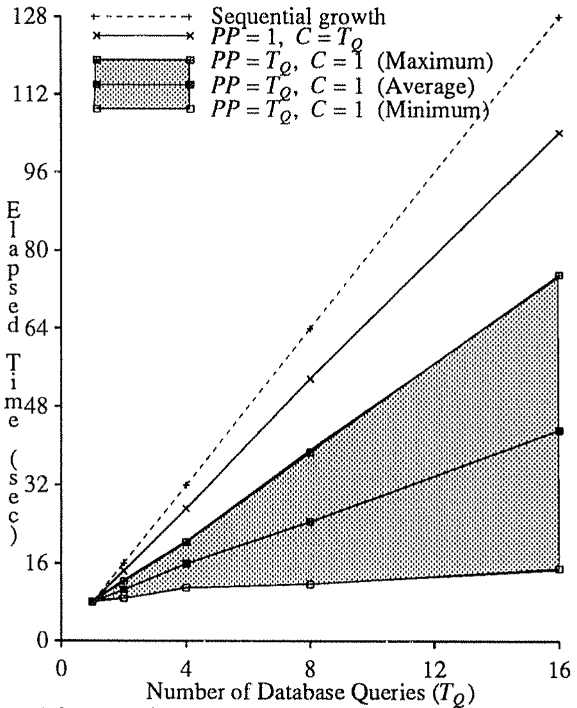

Figure 5.1: Elapsed time of Prolog processes with multiple database queries and a single Loader.

We observe that, in all cases in the examined range, the system takes advantage of parallelism between IDM and the rest of BERMUDA. For any value of T_Q, queries are executed in less time than if they were executed sequentially. As expected, the case of a single Prolog process can take less benefit of that than the case of multiple Prolog processes. This is because of the specific ordering that is imposed on processing the result tuples of each query based on the position of the corresponding database clusters in the Prolog clause. No such restriction exists for the case of multiple Prolog processes, where the decrease in the average elapsed time is rather significant.

Another interesting observation is that all diagrams are approximately linear. This is because we presented results with a single Loader, which effectively serializes the query execution in IDM and, therefore, linearizes the overall behavior.

Figure 5.2 sheds some light on the other source of potential parallelism that exists in the system, i.e., that of concurrent execution of queries in IDM. It shows the elapsed time of Prolog processes as a function of the number of available Loaders. We should emphasize that multiple queries in IDM means concurrency in the query execution itself, but also means concurrent existence of multiple Loaders that fight for system resources at the same time. It is the effect of both phenomena that we observe in Figure 5.2. We present again two extreme examples for the case of sixteen total queries (T_Q=16), with intermediate cases having results that are between these two: eight Prolog processes, each one containing a single clause with two maximal database clusters (*PP*=8 and *C*=2), and a single Prolog process containing a single clause with sixteen maximal database clusters (*PP*=1 and *C*=16). We do not present the case of sixteen Prolog processes with a single database cluster in their clauses because no tests with more than two Loaders could be run with that set-up: the maximum number of concurrent processes allowed by Unix was exceeded (partly because of timing processes that we have used). As before, for the multiple Prolog process case, Figure 5.2 shows the range of elapsed times of the processes, as well as the average such time.

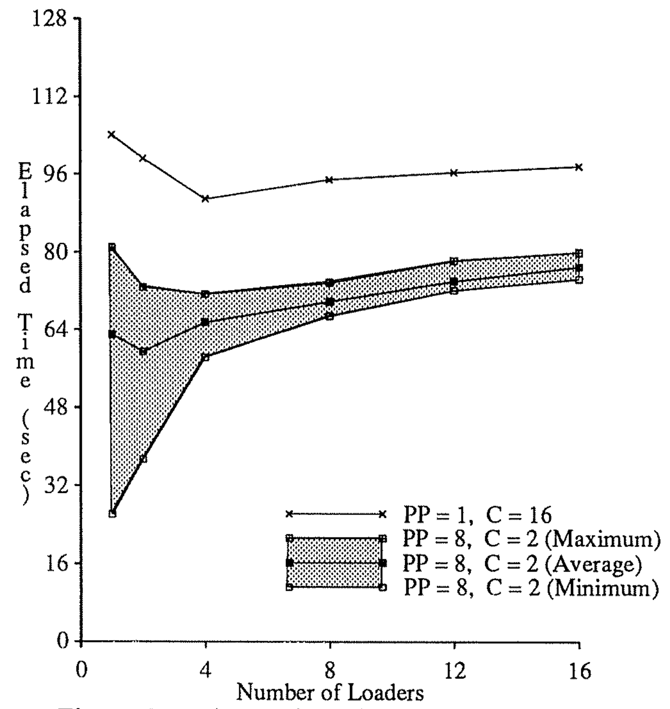

Figure 5.2: Elapsed time of Prolog processes as a function of the number of Loaders.

We observe that again, for the same reasons discussed above, many Prolog processes each one of which issues few queries can be processed faster than a single Prolog process that issues many queries. For the single Prolog case, the graph presents a minimum, indicating that some parallelism of queries in IDM is beneficial, but performance hurts beyond a certain point, due to resource and data contention. The differences, however, are relatively small. For the multiple Prolog case, the above still holds with respect to the average and the maximum elapsed time of a process, but not for the minimum one. The latter has a monotonically increasing value with the number of available Loaders. This has a rather intuitive explanation. With a single Loader, the query that is sent to it first is executed alone, without any competition for resources. The higher the number of available Loaders, the higher the competition for the queries that are sent to them in the beginning, and therefore, the higher the delay for the first one to finish. Interestingly, in the complete set of experiments that we have run, we have observed that as *PP* decreases and *C* increases, all diagrams, including the one of the minimum elapsed time, start resembling the diagram of the single Prolog process, i.e., they present a minimum. That is, as the elapsed time of a process increases, the optimum amount of parallelism increases as well. In some sense, this is also seen in Figure 5.2, since for the fastest process, the optimum is *L*=1, for the average process, the optimum is *L*=2, and for the slowest process, the optimum is *L*=4.

Another interesting point that emerges from Figure 5.2 is that the more parallelism that exists in the system, the more predictable the cost of concurrent queries becomes, i.e., the smaller the difference between the fastest and the slowest Prolog process. This is clearly seen for the cases with more than eight Loaders, where that difference is minimal. Intuitively, this is explained as follows. More Loaders imply higher contention, which delays all processes for a longer period of time. During that time, all processes progress further in their execution, and therefore, as some of them start finishing, the remaining ones are closer to completion as well, and finish soon after that.

The above results are helpful in deciding the optimal value of *L*. Clearly, the specific value depends on the types of queries that are run in the system. The trends, however, for the same amount of total load are as follows: (a) the fewer Prolog processes that are active, the more Loaders that should be used; (b) the higher the elapsed time that is to be optimized (from minimum to average to maximum), the more Loaders that should be used; (c) the higher the desired predictability of the elapsed times of processes, the more Loaders that should be used. Ideally, the system should be dynamically deciding on the number of Loaders to be used in each case based on the above criteria.

6. CRITIQUE

In building and experimenting with BERMUDA, we have gained valuable experience on how loosely-coupled systems of its kind should be developed. There are several of our decisions that we would revise if we were to build BERMUDA again. In addition, there are several features that we believe would be very useful if they were incorporated in the system. The following subsections discuss the points where BERMUDA could be improved together with our suggestions for how this could be done. Specifically, we address some of the potential functionality enhancements of the system, some design alternatives and their expected performance, and some implementation improvements.

6.1. Functionality Enhancements

We discuss three functionality enhancements of the current version of BERMUDA, two with respect to forming database clusters and one with respect to the use of cached query answers..

In most cases, the larger (and fewer) the maximal database clusters are, the more performance benefits from the query processing and optimization techniques of the database system. In that respect, the current version of BERMUDA is rather restrictive in its definition of a database cluster. One very helpful extension would be to form database clusters even from predicates that do not appear consecutively in a Prolog clause, as long as the semantics of Prolog, modulo our assumptions of Section 2, is retained. As an example, consider the following clause:

$$\mathbf{p}_1 \text{ :- } \cdots, \mathbf{p}_2, \mathbf{d}_1, \mathbf{p}_3, \mathbf{d}_2, \mathbf{p}_4, \cdots .$$

It would be nice if "$\mathbf{d}_1,\mathbf{d}_2$" could form a database cluster. This is not always possible, and it depends on the definition of $\mathbf{p}_3$. Abstractly, the question is whether two consecutive predicates in a Prolog clause can be swapped yielding an equivalent program, i.e., retaining the program semantics, or not. The problem appears to be quite complex in its full generality, but we suspect that there are several frequent cases that can be detected with little effort.

Another weakness of the system is with respect to what constitutes a database query. All database queries are formed from predicates in the same clause. That is, the preprocessor does not examine the program globally to replace nondatabase predicates with their definitions if those happen to involve database predicates only. Also, no negation is included in database queries due to their occasional semantic problems. Such enhancements will be very useful to BERMUDA and have been incorporated in other systems [Deno86]. They can be realized by employing techniques from partial evaluation of Prolog programs [Venk84], but they will also make the preprocessor significantly more expensive.

In the current design of BERMUDA, the cache is used to answer identical queries that are repeated. Similarly to other systems [Ceri86, Ceri89a], this can be expanded so that the BA checks for *query subsumption* [Fink82] and not for query equivalence only. This will increase the number of times that database queries are not sent to IDM. It will also increase, however, the complexity of the BA, because the latter will need to operate on flat files to obtain the answer of the subsumed query. In addition, since the unit of transformation of Prolog subclauses to database queries is the maximal database cluster in BERMUDA, we feel that query subsumption will not be applicable as often as in other systems whose corresponding unit was a single database predicate [Ceri86, Ceri89a].

6.2. Design Alternatives

In this subsection, we identify some alternative designs for BERMUDA and briefly discuss some estimates of their overhead. The primary differences among these alternatives are in the process structure of the system. These differences, of course, affect the functionality provided to the user as well as performance. These alternatives that we want to discuss are pictorially shown in Figure 6.1. For each alternative, Figure 6.1 shows a simple path from Prolog to IDM that includes all intermediate modules (if any). In reality, some of these modules, e.g., Prolog processes or Loaders, may be replicated, but this aspect is unrelated to the interests of this subsection.

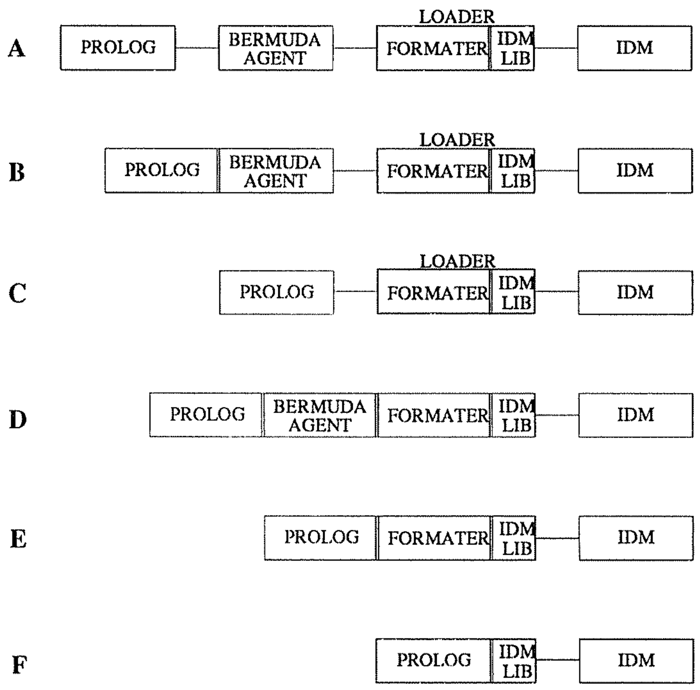

Figure 6.1: Design alternatives for BERMUDA.

Alternative **A** is the one that has been adopted in BERMUDA (see Figure 2.1). Alternative **B** has one less process than **A**, by making the BA a library that is called from within Prolog. Presumably, each Prolog process will have dedicated Loaders to it that are created when the first database query is issued and are destroyed at the end of the session. Otherwise, the BA code should be enhanced to poll the Loaders for availability. Compared to **A**, the main loss in functionality in this alternative is that cached answers and query templates are not shared across Prolog processes. Also, opportunities for prefetching are lost, and the system will need to rely on the operating system for that.

Alternative **C** abandons sharing altogether and the whole module of the BA is removed. There is no external caching in this alternative, so data is communicated from the Loader to the Prolog process through a pipe. Caching of query answers, however, can be done in the Prolog address space. Otherwise, all queries, whether they have been posed before or not, and whether they conform to a template that has been repeated or not, are sent to IDM. In this alternative, each Prolog process will need to have dedicated Loaders.

Alternative **D** does allow sharing among queries from the same Prolog process, but implements the entire system in a single process. The big disadvantage of doing so is that processing has to stop whenever a tuple is requested from IDM. Unlike alternative **C**, since external caching is used in this alternative, every page in the answer should be written in a file.

Alternative **E** is the most spartan one, offering the least functionality possible, essentially only translating queries into SQL and formating the answers back according to the needs of Prolog. This is accomplished by the routines that constitute the Formater. The blocking problems of the previous alternative are shared by this one as well.

Finally, alternative **F** is not realizable, since there are differences between Prolog and IDM that prohibit them from communicating directly. It appears in Figure 6.1, however, because it serves as a basis for comparison of all the others.

In the experiments that have been conducted to measure the overhead of the Connector Module of BERMUDA (Section 4.2), enough data has been collected for all parts of the system to derive some estimates for the corresponding overhead of all the alternatives mentioned above. We should emphasize that these estimates are very approximate, and accurate numbers cannot be obtained unless all these alternatives are implemented. They do give, however, some indication of the cost of the various services in BERMUDA and are useful for that. What constitutes the Connector Module is different in each alternative. As a general definition, we assume that the Connector Module consists of everything between Prolog and the IDM library, e.g., alternative **F** does not have a Connector Module.

To be consistent with the numbers presented in Section 4.2, the results shown below represent the overhead paid in the Connector Module per page of the query answer. For each alternative **X**, that overhead is denoted by OV_X and has been calculated as follows. For alternative **A**, the overhead is the one presented in Section 4.2. We use the average of the observed overheads that are shown in Table 4.5. For alternative **B**, the overhead is equal to OV_A minus the cost of a write to a pipe, the cost of a read from a pipe, and the cost of a context switch. The cost of this entire sequence of actions has been measured directly in BERMUDA. For alternative **C**, there is even higher difficulty in obtaining its overhead with some accuracy. To a first approximation, however, we can assume that it is related to alternative **E** in the same way that **A** is related to **B**, since in both cases, the difference is that of a write to a pipe, a read from a pipe, and a context switch. That is, the equality $OV_A - OV_B = OV_C - OV_E$ is assumed to hold. For alternative **D**, the overhead is equal to the CPU time of the BA and the Formater, for which direct measurements have been taken in BERMUDA. For alternative **E**, the overhead is equal to the CPU time of the Formater alone. Finally, alternative **F**, the basis for comparison, has zero overhead. Table 6.1 contains the estimates for the overhead of all alternatives computed as described above. All numbers in the second column are in msec and represent averages of several measurements, with very small variance among them. The third column indicates the ratio of the overhead of each alternative over the overhead of **A**.

Design alternative	Connector Module overhead	
	msec/page	% of A
A	117	100
B	69	59
C	53	45
D	30	26
E	5	14
F	0	0

Table 6.1: Overhead of Connector Module for design alternatives **A** - **F**.

We should emphasize that a lower estimated overhead for some alternative does not necessarily imply that the response time of queries will be lower as well. This is especially true for alternatives **D** and **E** where every request to IDM results in blocking of all other useful processing.

Using the estimates in Table 6.1, we can approximately calculate the reduction in the overhead of BERMUDA (alternative **A**) if the various services that are offered by it are abandoned as follows. If the ability to share query answers and query templates across multiple Prolog processes is abandoned (as in **B**) then the overhead is reduced by approximately 41%. If the ability to share is abandoned altogether (as in **C**) then the overhead is reduced by approximately 55%. If the non-blocking interface to IDM is abandoned (as in **D**) then the overhead is reduced by 74%. Finally, if both services are abandoned together (as in **E**) then the overhead is reduced by 86%. Note that it is impossible to accurately assign percentages to each service independently because each one is affected by the existence of the other.

6.3. Implementation Improvements

We discuss several aspects of the implementation of BERMUDA that are suboptimal. These deal with the scheduling policy for queries that wait for Loaders to become available, the synchronization between the BA and the Loaders, the buffer management of the BA, the BA-Prolog and the BA-Loader communication, and the caching of query answers.

For queries in the queue waiting for Loaders, there are cases where a more sophisticated scheduling policy than First-Come-First-Served is appropriate. For example, consider the following clause:

$$p_1 :- \cdots, p_2, \mathbf{d}_1, p_3, \mathbf{d}_2, p_4, \cdots .$$

This will generate two queries, namely "$\mathbf{d}_1$" and "$\mathbf{d}_2$". It is conceivable that at some point the answer for "$\mathbf{d}_1$" is partially computed, a $\mathbf{d}_1$ page has been passed to Prolog, the query "$\mathbf{d}_2$" has been sent, and all of the Loaders are busy. In this case, it makes more sense for "$\mathbf{d}_2$" to preempt the processing of "$\mathbf{d}_1$" and take over its Loader, as the complete answer of the latter will be requested by Prolog before any other tuple of the former is.

The above preemptive policy can be implemented if the same process is allowed to have multiple database queries open simultaneously. In that case, when a Loader asks for a tuple from one of the queries, it has to block, but after it receives the answer, it can ask for a tuple from another query. IDM stops consuming resources on behalf of the first query and records its status so that processing can resume later. Allowing a Loader to deal with multiple open queries simultaneously would complicate its design significantly. Every database query from the same clause of the same Prolog process would have to be sent to the same Loader. The protocol should be enhanced to accommodate for such information to be propagated. The Loaders would need to be in continuous contact with the BA in case multiple queries are to be assigned to them. The BA would become more complicated as well, especially to address the case of multiple instances of the same query being open at the same time. The above enhancement, however, should have a positive effect on the overall performance.

The above feature of stopping the processing of some query and starting another one can be used to improve the treatment of cuts (!) as well. For queries that appear before a !, processing should stop after a few tuples have been generated and sent to Prolog. If one of them satisfies the query, all remaining ones are useless, and the query can be aborted. With such a scheme, much work that IDM and the Loader must do in the current implementation can be saved. For the case of cuts, of course, the ideal would be to send a query that asks for a single tuple, which would require for the query language to support nondeterministic choice. With the exception of LDL [Naqv89], however, languages of current database systems do not support that.

Regarding the buffer replacement strategy that was used in the BA, simplicity was the primary motivation for our choice. A more complicated strategy, like the one used in the first version of BERMUDA [Ioan88], would definitely be beneficial. However, disk I/O is a very small fraction of the current overhead due to prefetching. This has been an additional reason for the decision to abandon the original, complex buffer management strategy and adopt the current one.

One peculiar aspect of our implementation is the difference between the BA-Prolog and the BA-Loader connections with respect to the way query answers are transferred between processes. In the former, the BA reads from a file and uses a pipe to send a page to Prolog, whereas in the latter, the Loader writes to a file and the BA simply reads from it. There is no reason why Prolog and the BA cannot use a shared file for this purpose as well. Such a scheme would definitely have less overhead than the current one, because Prolog and the BA would not need to communicate for every page of the answer file, the number of context switches would be reduced, and only a direct file read (probably sped up by prefetching of the operating system) would be necessary instead of a file read, a pipe write, and a pipe read. The disadvantages of such a scheme are relatively minor: no specialized prefetching or buffering strategies can be used, these tasks being left to the operating system; servicing remote Prolog processes requires the availability of a remote file system, e.g., NFS or RFS, on the BA or the Prolog machine. If we were to implement BERMUDA again, we would choose a shared file scheme. The reason for us choosing the scheme that we have implemented is historical. In the first version of Bermuda, the BA was passing individual tuples to Prolog processes instead of pages (as opposed to the BA and the Loaders that used pages for their communication from the beginning). Specialized prefetching and buffering techniques were much more crucial at the time and required the centralized control that the BA offered. Thus, reading of the answer file was implemented in the BA.

For both the BA-Prolog and the BA-Loader connections, a shared memory communication scheme would be even better than the shared file one, because then much I/O would be avoided. For operating systems that support shared memory between processes, this would be the scheme of choice [Baro86]. Similarly, the Loaders could be abandoned altogether if the operating system supported light-weight processes (threads) [Drav87]. In the case of BERMUDA, however, which was developed under Unix 4.3 BSD, since the latter does not offer such services, a shared file scheme appears to be the preferable choice.

Finally, as we have mentioned above, if sharing of query answers across multiple Prolog processes is of no interst, then such answers can be cached into the virtual memory of Prolog. For several cases, especially when the total amount of cached results is not very large, cache hits will have better performance in that scheme. This is because, in that case, no process boundary would be crossed and also most likely the requested data will be resident in main memory and I/O will be avoided.

7. CONCLUSIONS

We have described the design and implementation of BERMUDA, which is a loosely-coupled system that interfaces possibly several Prolog processes to the Britton-Lee Intelligent Database Machine (IDM-500). We have also discussed the performance results of several types of experiments with BERMUDA. Except for simple queries, e.g., selections, BERMUDA appears to be more efficient than Prolog, and its performance is close to that of IDM.

Caching of query answers and query templates improves the overall performance greatly. By taking a profile of the system performance, we have also calculated the overhead of the Connector Module through indirect measurements. The obtained estimates indicate that this overhead is relatively small. Finally, when multiple Prolog processes are concurrently active, BERMUDA can take advantage of parallelism among its modules and increase throughput.

In the previous section, we have discussed several aspects of BERMUDA that could be modified for improved functionality and performance. Even with the current version of the system, however, we feel that the competitiveness of loosely-coupled systems has been demonstrated. We believe that loose coupling of systems will be increasingly necessary in the diversely heterogeneous environments of the future. We hope that the conclusions of our work on BERMUDA will help others in similar efforts of building such systems.

8. REFERENCES

[Astr76]
Astrahan, M. et al., "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems* **1**, 2 (June 1976), pp. 97-137.

[Baro86]
Baron, R. et al., "Mach: A New Kernel Foundation For UNIX Development", in *Proc. Winter '86 USENIX*, 1986, pp. 93-112.

[Bitt83]
Bitton, D., D. J. DeWitt, and C. Turbyfill, "Benchmarking Database Systems: A Systematic Approach", in *Proc. 9th International VLDB Conference* , Florence, Italy, August 1983, pp. 8-19.

[Bocc86a]
Bocca, J., "On the Evaluation Strategy of EDUCE", in *Proc. of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pp. 368-378.

[Bocc86b]
Bocca, J., "EDUCE - A Marriage of Convenience: Prolog and a Relational DBMS", in *Proc. of the 1986 Symposium on Logic Programming*, Salt Lake City, UT, September 1986, pp. 36-45.

[Bocc86c]
Bocca, J. et al., "Some Steps towards DBMS Based KBMS", in *Information Processing 86*, North Holland, September 1986, pp. 1061-1067.

[Ceri86]
Ceri, S., G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently", in *Proc. of the 1st International Conference on Expert Database Systems*, Charleston, SC, April 1986, pp. 141-153.

[Ceri89a]
Ceri, S., G. Gottlob, and G. Wiederhold, "Efficient Database Access from Prolog", *IEEE Transactions on Software Engineering* **15**, 2 (February 1989), pp. 153-164.

[Ceri89b]
Ceri, S., G. Gottlob, and L. Tanca, "What you Always Wanted to Know about Datalog (and Never Dared to Ask)", *IEEE Transactions on Knowledge and Data Engineering* **1**, 1 (March 1989), pp. 146-166.

[Chan86]
Chang, C. L. and A. Walker, "PROSQL: A Prolog Programming Interface with SQL/DS", in *Expert Database Systems, Proc. from the First International Workshop*, edited by L. Kerschberg, Benjamin/Cummings, Inc., Menlo Park, CA, 1986, pp. 233-246.

[Deno86]
Denoel, E., D. Roelants, and M. Vauclair, "Query Translation for Coupling Prolog with a Relational Database Management System", in *Proc. Workshop on Integration of Logic Programming and Databases*, Venice, Italy, December 1986.

[Drav87]
Draves, R. P. and E. C. Cooper, *"C Threads"*, Technical Report, Dept. of Computer Science, Carnegie Mellon

University, 1987.

[Fink82]
Finkelstein, S., "Common Expression Analysis in Database Applications", in *Proc. of the 1982 ACM-SIGMOD Conference on the Management of Data*, Orlando, FL, June 1982, pp. 235-245.

[Gall78]
Gallaire, H. and J. Minker, *Logic and Data Bases*, Plenum Press, New York, N.Y., 1978.

[Ghos88]
Ghosh, S., C. C. Lin, and T. Sellis, Implementation of a Prolog-INGRES Interface, *ACM-SIGMOD record* **17**, 2 (June 1988), pp. 77-88.

[Ioan88]
Ioannidis, Y. E., J. Chen, M. A. Friedman, and M. Tsangaris, BERMUDA - An Architectural Perspective on Interfacing Prolog to a Database Machine", in *Proc. of the 2st International Conference on Expert Database Systems*, Tysons Corner, VA, April 1988, pp. 91-105.

[Jark84]
Jarke, M., J. Clifford, and Y. Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System", in *Proc. of the 1984 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1984, pp. 296-306.

[Morr86]
Morris, K., J. D. Ullman, and A. VanGelder, "NAIL! System Design Overview", in *Proc. of the 3rd International Conference on Logic Programming*, London, England, July 1986, pp. 554-568.

[Nais83]
Naish, L. and J. A. Thom, *"The MU-Prolog Deductive Database"*, Technical Report 83/10, Computer Science Dept., University of Melburne, November 1983.

[Naqv89]
Naqvi, S. and S. Tsur, *A Logical Language for Data and Knowledge Bases*, Computer Science Press, New York, NY, 1989.

[Nico83]
Nicolas, J. M. and K. Yazdanian, "An Outline of BDGEN: A Deductive DBMS", in *Information Processing 83*, edited by R. E. Mason, North Holland, 1983, pp. 711-717.

[Ritc78]
Ritchie, D. M. and K. Thompson, "The UNIX Time-Sharing System", *The Bell System Technical Journal* **57**, 6 (July-August 1978), pp. 1905-1929.

[Seli79]
Selinger, P. et al., "Access Path Selection in a Relational Data Base System", in *Proc. of the 1979 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1979, pp. 23-34.

[Ston85]
Stonebraker, M., "Triggers and Inference in Data Base Systems", in *Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Islamorada, FL, February 1985.

[Ubel84]
Ubell, M., "The Intelligent Database Machine (IDM)", in *Query Processing in Database Systems*, edited by W. Kim, D. Reiner, and D. Batory, Springer-Verlag, New York, N.Y., 1984.

[Venk84]
Venken, R., "A Prolog Meta-Interpreter for Partial Evaluation and its Applications to Source to Source Transformation and Query-Optimization", in *Proc. ECCAI*, edited by T. O'Shea, Elsevier (North Holland), Amsterdam, The Netherlands, 1984, pp. 91-100.

[Will87]
Williams, M. H., P. Massey, and J. Crammond, *"Benchmarking Prolog for Database Applications"*, Technical Report No. 87/7, Heriot-Watt University, Edinburgh, 1987.