A Data/Knowledge Base Management Testbed and Experimental Results on Data/Knowledge Base Query and Update Processing

Raja Ramnarayan, Hongjun Lu

Honeywell Corporate Systems Development Division
1000 Boone Avenue North, Golden Valley, MN 55427

and

National University of Singapore
Department of Information and Computer Science
Lower Kent Ridge, Singapore 0511

Abstract

This paper presents our experience in designing and implementing a data/knowledge base management testbed The testbed consists of two layers, the knowledge manager and the database management system, with the former at the top The testbed is based on the logic programming paradigm, wherein data, knowledge, and queries are all expressed as Horn clauses The knowledge manager compiles pure, function-free Horn clause queries into embedded-SQL programs, which are executed by the database management system to produce the query results The database management system is a commercial relational database system and provides storage for both rules and facts First, the testbed architecture and major data structures and algorithms are described Then, several preliminary tests conducted using the current version of the testbed and the conclusions from the test results are presented The principal contributions of this work have been to unify various concepts, both previously published and new ones we developed, into a real system and to present several insights into data/knowledge base management system design gleaned from the test results and our design and implementation experience

1 Introduction

There has been extensive research on integrating database and artificial intelligence technologies to develop a new generation of data/knowledge base management systems (D/KBMSs) or expert database systems A number of important technical issues have been identified and solutions and algorithms for resolving them proposed

Our objective was to build a testbed to experimentally investigate the major issues in designing and implementing a high performance D/KBMS, including design methodology, system architecture, and query optimization and processing techniques This paper presents our experience in designing and implementing such a testbed

Although a large number of papers addressing various issues in D/KBMS design have been published [1, 2, 3], our testbed is one of the first attempts to implement these ideas in a real system This is a major contribution of our work In addition, we have finalized many details of published algorithms and developed a number of algorithms to fill the gaps between them

The testbed is based on the logic programming paradigm, wherein data, knowledge, and queries are all expressed as Horn clauses Our overall approach to designing the testbed has been the so called compilation approach, wherein rules relevant to the query are first extracted from the knowledge base, the order of evaluating them determined, and optimizations applied The output of the compilation process is a program segment, which is compiled and linked with a run-time library containing functions accessing the database and functions implementing recursive query processing algorithms The query results are determined by executing the object code thus generated

We have conducted several preliminary experiments using the current version of the testbed This paper presents the results of the tests, an analysis of the results, and conclusions drawn from them The insights into D/KBMS design gleaned from the test results are another key contribution of our work The testbed has proven very useful for experimentation and we feel that it is a valuable tool for further investigation of technical issues in D/KBMS design

The remainder of this paper is organized as follows Section 2 presents background concepts germane to data/knowledge base (D/KB) query processing Section 3 describes the testbed architecture Section 4 presents some major data structures and algorithms developed for the testbed Section 5 presents some preliminary experiments and results Section 6 presents conclusions drawn from the test results and our design and implementation experience

2 Background Concepts in D/KB Query Processing

2.1 Data/Knowledge Base

The D/KB consists of a set of Horn clauses and schemas A Horn clause has the form

$$head \leftarrow body$$

where *head* is zero or one atomic formulas (predicates with arguments supplied) and *body* is a conjunction of zero or more atomic formulas All variable arguments are implicitly universally quantified Figure 1 shows a sample set of Horn clauses

$$\begin{aligned}
R_1 &\quad p(X, Y) \leftarrow p_1(X, Z), q(Z, Y) \\
R_2 &\quad p(X, Y) \leftarrow b_3(X, Y) \\
R_3 &\quad p_1(X, Y) \leftarrow b_1(X, Z), p_1(Z, Y) \\
R_4 &\quad p_1(X, Y) \leftarrow b_4(X, Y) \\
R_5 &\quad p_2(X, Y) \leftarrow b_2(X, Z), p_2(Z, Y) \\
R_6 &\quad p_2(X, Y) \leftarrow b_5(X, Y) \\
R_7 &\quad q(X, Y) \leftarrow p(X, Z), p_2(Z, Y)
\end{aligned}$$

Figure 1 Sample data/knowledge base

The logical interpretation of a Horn clause is that the body *implies* the head For example, $R1$ in Figure 1 means that for all X, Y, and Z $p_1(X, Z)$ and $q(Z, Y)$ implies $p(X, Y)$ A *relation definition* is the set of clauses whose head refers to a given relation

Horn clauses can be further divided into two groups *facts* and *rule* A *fact* is a Horn clause with an empty body and no variables in its head Facts may be written with no implication sign e g , *parent('john 'mary')*, which says "*john*" is the parent of "*mary*" A *rule* is a Horn clause that is not a fact All Horn clauses in Figure 1 are rules Predicates defined by facts alone are called *base predicates* while those defined by rules alone or by rules and facts are called *derived predicates* In Figure 1, the b_i's are base predicates, while p q p_1, and p_2 are derived predicates

We can assume without loss of generality that a predicate is defined entirely by rules or entirely by facts If a set of Horn clauses does not meet this condition, it can easily be transformed into a set of clauses that does For example, the following two sets of Horn clauses are equivalent

Set1	Set2
$p(X, Y) \leftarrow a(X, Z), b(Z, Y)$	$p(X, Y) \leftarrow a(X, Z), b(Z, Y)$
$p(a, b)$	$p(X, Y) \leftarrow p_1(X, Y)$
$p(c, d)$	$p_1(a, b)$
	$p_1(c, d)$

Thus, a D/KB can be partitioned into rule relations and fact relations The set of rule relations is called the *intensional data base*, or *rule base*, while the set of fact relations is called the extensional data base, or *data base* The intensional data base contains only derived predicates and the extensional data base only base predicates

The motivation for distinguishing between the intensional and extensional knowledge bases is that relations defining base predicates are stored

This work was supported in part by Rome Air Development Center under the Very Large Parallel Data Flow program contract number F30602 85-C 0215 The technical contract monitor for this program is Dr Raymond Liuzzi

as traditional database relations, while those defining derived predicates are stored in some compiled form to permit efficient access during query processing

2 2 Recursion

This section introduces concepts pertaining to recursive query processing, which is a key concept differentiating traditional database systems from D/KBMSs

A predicate q is said *reachable* from a predicate p if

(i) q is in the body of a rule having p as its head, or

(ii) q is in the body of a rule having s as its head and s is reachable from p

For example, in Figure 1, p_1 is reachable from p So is b_1, because b_1 is reachable from p_1 and p_1 is reachable from p p_1 and p_2 are recursive predicates, while p and q are mutually recursive

A predicate p is *recursive* if it is reachable from itself Two or more derived predicates are *mutually recursive* if they are reachable from each other

A rule $p \leftarrow p_1, p_2, \ldots, p_n$ is called a *recursive rule* if at least one of the p_i s is mutually recursive to p Rules are *mutually recursive* if the predicates in their heads are mutually recursive Rules that are not recursive nor belong to a set of mutually recursive rules are called *exit rules*

The above concepts can be explained using a graph formalism The Predicate Connection Graph (PCG) [4] is representative of such a formalism Each node in a PCG represents a predicate Edges arise from rules If there is a rule of the form $p \leftarrow p_1, p_2, \ldots, p_n$, there is a directed edge from p to each of the p_i's Figure 2 shows the PCG for the sample set of Horn clauses for Figure 1

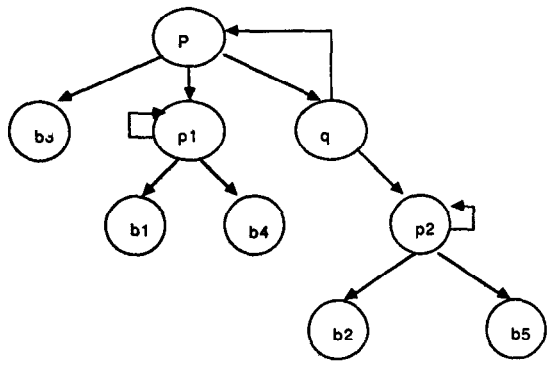

Figure 2 Predicate connection graph for rules in Figure 1

Mutually recursive predicates form the *strongly connected components* of a PCG, where a strongly connected component, also called a *clique*, of a graph is a set of nodes in which there is a directed path between each pair of nodes

In the context of D/KB query processing, we will use a somewhat broader definition of cliques Here, by clique we will mean a set of recursive predicates as well as the rules defining these predicates In general, some of these rules are recursive, others form mutually recursive groups, and the rest are exit rules For example, the cliques for the rule set in Figure 1 is shown in Figure 3

Clique	Recursive predicates	Recursive rules	Exit rules
1	p, q	R_1, R_7	R_2
2	p_1	R_3	R_4
3	p_2	R_5	R_6

Figure 3 Cliques for rules in Figure 1

Cliques actually represent a partitioning of the set of rules into disjoint groups, the significance of this partitioning being that predicates in the same clique must be evaluated as a whole using the rules in that clique Also, there must be at least one exit rule for each clique, to ensure that the evaluation of the predicates terminates

2 3 D/KB Queries and Evaluation Graph

D/KB queries are also expressed as Horn clauses For example, a typical query against the rule set in Figure 1 is

$$query(X) \leftarrow p(X, \ john)$$

Evaluating the above D/KB query is tantamount to evaluating the derived predicate *query*, which in turn involves evaluating all the predicates reachable from *query* The predicates reachable from *query* are called *relevant predicates*, while the rules defining them are called *relevant rules* Evaluating D/KB queries is then tantamount to evaluating the relevant predicates using the relevant rules

As described in the previous section, the relevant rules can be partitioned into cliques and from this partitioning we can construct a new graph, called the *evaluation graph* for the query This is a directed graph whose nodes are either cliques or non-recursive derived predicates Clique nodes can be thought of as the strongly connected components of the PCG collapsed into single nodes, while non-recursive predicate nodes are the same as those in the PCG There are four types of edges in the evaluation graph (1) $P \rightarrow C$ indicates that some predicate in the clique C appears in the body of a rule defining P, (2) $C \rightarrow P$ indicates that P appears in the body of a rule defining some predicate of C, (3) $P_1 \rightarrow P_2$ indicates that P_2 appears in the body of a rule defining P_1, and (4) $C_1 \rightarrow C_2$ indicates that some predicate of C_2 appears in the body of a rule defining some predicate of C_1 While the PCG may be a cyclic graph, the evaluation graph is acyclic It is then possible to order the nodes in this graph in such a way that all predicates reachable from a predicate p appear as predecessors of p some of these predecessors being clique nodes containing some of the reachable predicates We call the total order with this property the *evaluation order list* The evaluation order list specifies the order in which to evaluate the cliques and non recursive derived predicates during D/KB query processing Figure 4 is the evaluation graph obtained after we insert the above query into the rule set in Figure 1

Usually, there is more than one possible evaluation order list for a query For example, from the evaluation graph in Figure 4, we can obtain two evaluation order lists $\{C_2, C_3, C_1, query\}$ and $\{C_3, C_2, C_1, query\}$ The choice of evaluation order list is one of the optimization problems in D/KB query processing that we have not addressed yet

2 4 Evaluating Predicates and Cliques

There are essentially two strategies for evaluating predicates during D/KB query processing – *top down evaluation* and *bottom up evaluation*. Bottom-up evaluation starts with the base predicates in the body of the relevant rules and keeps combining them with other predicates in the body to produce the head predicates till the predicate *query* is generated Top-down evaluation starts with *query* and keeps evaluating predicates in the body of the relevant rules by propagating the bindings in the head predicates of these rules

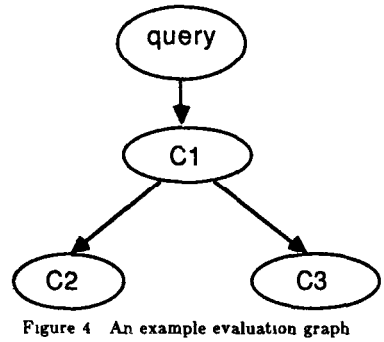

Figure 4 An example evaluation graph

In general, some of the relevant predicates will be non-recursive, with the rest being partitioned into cliques as explained in section 2 2 Bottom-up evaluation of non-recursive predicates is equivalent to computing a relational algebra expression On the other hand, bottom-up evaluation of a clique of recursive predicates is equivalent to computing the least fixed point (LFP) of a set of recursive relational algebra equations of the form, $r_i = f_i(r_1, \ldots, r_n)$, $i = 1, \ldots, n$, where the f_i s are relational algebra expressions

Examples of top-down evaluation are the Henschen-Naqvi method [5] and Prolog, while naive evaluation and semi-naive evaluation [6] are examples of bottom-up evaluation

2 5 Optimisation

Bottom up strategies are simpler and easy to implement, but they compute a lot of useless results, since they do not use knowledge about the query (i e constants in the query rule) to restrict the search space as top-down strategies do Several strategies have been proposed for use with bottom up strategies to improve performance, e g , magic sets [7], supplementary magic sets [8], counting and supplementary counting [9] The main idea behind these strategies is the use of *sideways information passing (sip)* to propagate the information in the query (query constants) and restrict the computation to tuples that are related to the query Beeri and Ramakrishnan have developed a uniform framework to describe and compare these strategies and to understand the basic ideas that are common to them [10]

The common feature of the above optimization strategies is that they are all rule rewriting systems That is, they rewrite the original set of rules into an equivalent one whose evaluation is expected to be more efficient than that of the original

3 Testbed Architecture

The D/KBMS architecture in the compilation approach can be functionally divided into two layers

- Knowledge Manager (KM), which compiles intensional database queries, using appropriate optimizations into a program that accesses the extensional database and
- Data Base Management System (DBMS), which executes the program generated by the Knowledge Manager

This two layered architecture is really a reference architecture D KBMSs with widely varying performance are realized via different choices for the KM optimization and LFP evaluation strategies and the KM/DBMS interface

We have designed and implemented a data/knowledge base management testbed using a commercial relational database system The Knowledge Manager in this testbed compiles pure, function-free Horn clause queries into embedded SQL programs, which are executed by the DBMS

Our objective was to build a tool that would serve as both a demonstration and performance measurement and evaluation platform As a demonstration platform, the testbed illustrates the motivation and basic functionality of a D/KBMS, the components of a D/KBMS architecture, alternative implementations of these components and their relative tradeoffs, and the factors contributing to D/KB query compilation and execution time As a performance measurement and evaluation platform, the testbed allows us to make quantitative performance measurements and to study system performance sensitivity and behavior with respect to several parameters

We point out that our testbed is not intended to be a high performance logic database system In particular, the choice of SQL as the KM/DBMS interface significantly degrades performance However, for the purposes listed above, the testbed has proven to be a very valuable tool

In a typical session with the testbed, the user enters a set of rules and facts These rules and facts are stored in a memory resident private environment called the *Workspace D/KB* The Workspace D/KB rules may refer to rules and facts stored on disk and the rules stored on disk may refer to rules and facts in the workspace The stored rules and facts constitute the *Stored D/KB* After entering a set of rules and facts into the Workspace D/KB, the user issues queries against them If he is satisfied that the rules and facts in the workspace D/KB are correct, he updates the stored D/KB with these rules and facts

The overall configuration of the testbed is shown in Figure 5

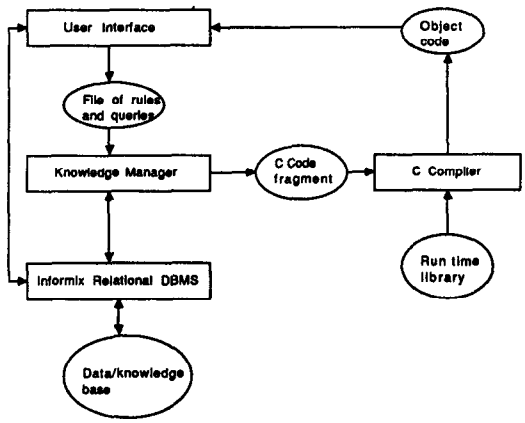

Figure 5 Testbed Architecture

The testbed consists of four components User Interface, Knowledge Manager, Run Time Library, and DBMS The User Interface handles interactions with the user The Knowledge Manager compiles D/KB queries into C program segments, which contain information about relevant rules and predicates and the evaluation order list for the query The program segment is then compiled and linked with the Run Time Library to generate the object code, which accesses the data/knowledge base and returns the query results The Knowledge Manager also handles updates to the Stored D/KB The DBMS is a commercial relational database management system with SQL and embedded SQL (in C) interfaces

3 1 User Interface

The main options provided by the User Interface are

- Enter rules, which allows the user to enter a set of Horn clauses into the workspace
- Enter query, which allows the user to enter and compile a Horn clause query
- Execute query, which allows the user to execute a previously compiled query
- Update stored D/KB, which allows the user to update the stored D/KB with rules and facts from the workspace

3 2 Knowledge Manager

The Knowledge Manager is the major component of the testbed It accepts Horn clauses and queries from the user interface and compiles queries into code fragments It consists of the following components Rule Parser, Workspace D/KB Manager, Stored D/KB Manager, Semantic Checker, Optimizer, and Code Generator The KM architecture is shown in Figure 6 The circles in this figure represent data structures and the boxes, components

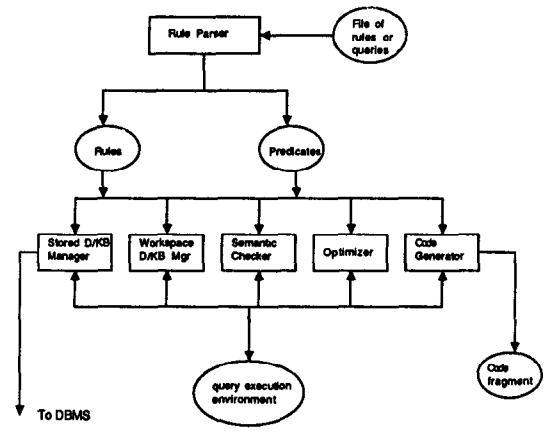

Figure 6 Knowledge Manager Architecture

3 2 1 Rule Parser

The Rule Parser parses the input Horn Clauses If no syntactic errors are detected, it generates an internal representation of the Horn clauses This representation consists of two hash tables, rules and predicates, into which information contained in the source form of the Horn clauses is loaded

3 2 2 Workspace D/KB Manager

The Workspace D/KB Manager provides the following functions

- Determine all predicates reachable from a given predicate
- Find the cliques in the workspace D/KB rules
- Generate the evaluation order list

3 2 3 Stored D/KB Manager

The Stored D/KB Manager provides the following functions

- Extract rules from the stored D/KB needed to solve a given set of predicates
- Read the intensional and extensional data dictionaries during type checking, which is the process where the types of the columns of the derived predicates are inferred (see the Semantic Checker, section 3 2 4 below)
- Update the stored D/KB with rules and facts from the Workspace D/KB

3 2 4 Semantic Checker

The testbed performs two kinds of semantic checks The first is to check for each derived predicate reachable from the query, whether there is a rule defining it The second is a type check for inferring the types of the derived predicates and also checking whether the same types are inferred from all the rules defining a predicate

3 2 5 Optimizer

D/KB query processing performance strongly depends upon the efficiency of LFP computation The Optimizer uses the generalized magic sets algorithm proposed by Beeri and Ramakrishnan [10] to rewrite the relevant rules into a new set of rules that is equivalent to the original set but whose LFP computation is more efficient

3 2 6 Code Generator

The code generated by the knowledge manager is actually a C program segment, which basically loads certain data structures in the object program with query specific information These data structures contain information similar to the nodes of the evaluation order graph For predicate nodes, the code fragment loads the predicate name, schema information (name and type of each column), and the SQL query to evaluate the body of each rule in which the predicate appears as head For clique nodes, the code fragment loads the same information, except that it differentiates between exit rules and recursive rules

3 3 Run Time Library

The Run Time Library currently contains our implementation of two bottom-up strategies for LFP evaluation naive evaluation and semi-

naive evaluation [6]. As we mentioned before, SQL is not the KM/DBMS interface to use when designing a high performance D/KBMS, since relational algebra cannot express LFP queries [1]. Consequently, the recursive equations are evaluated as an application program and there is not much scope for optimization here, several temporary tables are created and dropped during each iteration, which introduces a lot of overhead.

The code fragment produced by the Knowledge Manager is compiled and linked with the run time library to produce the object code, which is executed by the User Interface against the DBMS to give the query results.

3.4 Control Flow

This section describes the control flow among the components of the testbed during D/KB query compilation to give an overall picture of the system.

The User interface accepts user query and parses it.

The Workspace D/KB Manager determines all predicates reachable from the query. In general, there may be rules defining these predicates in the Stored D/KB. The Stored D/KB Manager extracts these rules by issuing appropriate SQL queries and loads them into the workspace.

At this point all the rules needed to solve the query (from both the Workspace and Stored D/KBs) are present in the workspace. The Workspace D/KB manager once again, determines all the predicates reachable from the query. This will now include workspace as well as stored D/KB predicates.

If optimization is to be used, the Optimizer is invoked to rewrite the rules relevant to the query. For the generalized magic sets strategy, it generates three sets of rules in the workspace: the adorned, magic and modified rules. It also generates an adorned version of the query.

The Workspace D/KB Manager then finds the cliques in the obtained rules and uses this information to generate the evaluation order list.

The Semantic Checker now performs the two tests.

The Code Generator generates the code fragment for each entry (clique or non-recursive predicate) in the evaluation order list.

The User Interface invokes compiling and linking procedures to compile the code fragment and link it with the run-time library to generate the object code.

The result of the query is returned to the user by running the object code.

4 Data Structures and Algorithms

The algorithms in the current version of the testbed can be grouped under four categories:

(i) Implementations of several previously published algorithms, after finalizing the details of a number of them. These include implementations of naive and semi-naive LFP evaluation as embedded-SQL programs, the latter using the differential approach described in [12].

(ii) New algorithms addressing specific issues in D/KB query processing. These include the type inferencing and checking algorithm and the algorithm for extracting the relevant rules from the Stored D/KB during compilation and updates.

(iii) Algorithms that unify various published concepts and new ones that we introduced. These include the D/KB query processing and update algorithms.

(iv) Algorithms that are designed but not yet implemented. These include an algorithm for generating a sideways information passing (sip) strategy as part of the generalized magic set optimization. Beeri and Ramakrishnan in their paper [10] just describe what a sip is but do not include an algorithm for generating one.

The details of these algorithms can be found in [13]. Here we briefly describe the intensional database storage structures, and the D/KB query processing and update algorithms.

4.1 Intensional Database Storage Structures

The DBMS stores both the extensional and intensional databases. Intensional database storage structure design is a key issue, since it influences the time to extract the relevant Stored D/KB rules during query processing. To speed up this time, the Stored D/KB rules should be stored in some compiled form, in addition to raw source form storage. The issue is what should be the compiled form storage structure?

Since the DBMS in our testbed is a commercial relational database system, the intensional database is stored as a set of relations. This set consists of four relations: *isystables*, *isyscolumns*, *irulesource* and *ireachablepreds*. The first two relations, *isystables* and *isyscolumns*, form the intensional database data dictionary. They contain the types of the columns of the derived predicates, which are derived from the base relations.

These two relations have the following schema:

isystables(*tablename char tableid integer*)

isyscolumns(*tableid integer*, *colname char*, *colnumber integer coltype integer*)

irulesource stores for each derived predicate p, the rules defining p. It has the following schema:

irulesource(*headpredname char rule char*)

ireachablepreds stores for each derived predicate p all the predicates reachable from p. The reachable predicates are obtained from the transitive closure of the PCG of the rules stored in *irulesource* and constitutes the compiled form of these rules. It has the following schema:

ireachablepreds(*frompredname char*, *topredname char*)

We use *ireachablepreds* as the compiled form of rules since it allows very efficient retrieval of the relevant rules from the stored D/KB. For example, retrieving all the rules needed to solve the query

$$query(X, Y) \leftarrow p(X, Z), q(Z, Y)$$

where p and q are derived predicates, is accomplished via the following SQL query:

SELECT *irulesource rule*
FROM *irulesource*, *ireachablepreds*
WHERE (*ireachablepreds topredname* = *irulesource headpredname*
OR *ireachablepreds frompredname* = *irulesource headpredname*)
AND (*ireachablepreds frompredname* = p
OR *ireachablepreds frompredname* = q)

This query retrieves all rules in the stored D/KB whose head predicates are either p, q, or the predicates reachable from p and q. To speed up the execution of this query, both *irulesource* and *ireachablepreds* are indexed.

4.2 D/KB Query Processing Algorithm

This section describes the D/KB query processing scenario implemented in our testbed.

1. Find the reachable predicate set P and the relevant rule set R for the query from both Workspace D/KB and Stored D/KB. This involves the following steps:
   - 1.1 Construct the PCG for the Workspace D/KB rules and the query.
   - 1.2 Initialize the reachable predicate set P to the predicates reachable from the query node in the PCG. Initialize the relevant rule set R to include rules in the Workspace D/KB whose head predicates are in P.
   - 1.3 Extract from the Stored D/KB all predicates reachable from the predicates in P and the rules whose head predicates are in P, using a query similar to the SQL query shown section 4.1. Insert them into P and R, respectively.
   - 1.4 Find new reachable predicates and relevant rules from the obtained rule set R and the Workspace D/KB, and insert them into P and R.
   - 1.5 Repeat steps 1.3 and 1.4 until no new rules and predicates are inserted into R and P.
2. Construct the PCG for the rule set R and find the cliques in the PCG.
3. Construct the evaluation graph and perform a topological sort of the evaluation graph to generate the evaluation order list.
4. Derive types for attributes of the derived predicates and perform semantic checks. If there is an error in either of the two semantic checks, do not perform the next step.
5. Evaluate the cliques and nonrecursive predicates as per the total order prescribed by the evaluation order list. If the node to be evaluated is a clique, optionally apply an optimization strategy to the clique and generate a new set of rules (e.g. magic set rules and modified rules). Evaluate the new set of rules. We say optionally above because our performance experiments indicate that there is a tradeoff in using optimization (see section 5). We have not yet implemented dynamic strategies for determining whether to apply optimization.

4.3 Stored D/KB Update Algorithm

This section describes how to update the Stored D/KB with rules from the Workspace D/KB. We point out that in our testbed only the intensional database storage structures are updated. In particular, there is no checking of these rules against any integrity constraints that may be associated with the Stored D/KB.

The D/KB update algorithm is as follows:

1. Let ΔDKB denote the workspace D/KB. Extract from the Stored D/KB all the rules needed to evaluate the derived predicates in ΔDKB. This can be done using a query similar to the SQL query shown in section 4.1. Let IDB_{rel} denote the extracted rules.
2. Construct the PCG of the rules in $\Delta DKB_{composite} = \Delta DKB \bigcup IDB_{rel}$, which denotes the set of rules which are either in ΔDKB or in IDB_{rel}.
3. Compute the transitive closure of this PCG. This gives all the predicates reachable from a given predicate in $\Delta DKB_{composite}$.
4. Perform the type checking algorithm which is described next in this section.
5. For each derived predicate p in $\Delta DKB_{composite}$, add tuples to *isystables* and *isyscolumns* if information on p is not present in these tables.

6 For each derived predicate p in $\Delta DKB_{composite}$ add tuples to *treachablepreds* by looking at the transitive closure computed in step 3

7 For each rule in ΔDKB add tuples to *trulesource*

We mentioned before that *treachablepreds* is the transitive closure of the PCG of the rules in *trulesource* The above algorithm computes this transitive closure incrementally That is, whenever the stored D/KB is to be updated, we do a transitive closure on only those portions of the stored D/KB that will be affected by the update (IDB_{rel}), and not of the entire stored D/KB This can result in substantial savings in update times for very large rule sets as the size of IDB_{rel} will be much smaller than that of the entire rule set

5 Measurement and Evaluation

We have conducted some preliminary experiments using the current version of the testbed The experiments are designed to quantitatively measure D/KBMS performance and to understand D/KBMS performance sensitivity and behavior with respect to various system parameters The basic motivation for doing these experiments is to experimentally investigate various issues related to the integration of knowledge management with database management systems, including the potential architecture of such systems, compilation techniques, and the effectiveness of major query optimization and processing methods This section describes these experiments and discusses the obtained results

5 1 Performance Measures and Parameters

The main D/KBMS performance measures we used are (i) D/KB query compilation time, t_c, (ii) D/KB query execution time, t_e, and (iii) D/KB update time t_u These measures are comprised of several components, which are discussed in the tests

The parameters that affect the above performance measures are grouped into three categories (i) D/KBMS architecture related parameters, (ii) workload related parameters, and (iii) D/KB query and update related parameters The D/KBMS architectural parameters relate to different aspects of the system architecture, the workload parameters relate to D/KB size, and the query and update parameters relate to the portions of the D/KB relevant to the query Table 1 describes these parameters

5 2 D/KB Characterization

The base relations used in the experimentation are all binary relations We characterize them in terms of their directed graph representation In this representation, a binary relation is represented as a directed graph, domain elements form the nodes of this graph and tuples the edges The following types of relations are used in the experimentation lists, full binary trees, directed acyclic graphs, and directed cyclic graphs The parameters in Table 2 are used to characterize these relations

We characterize the intensional database by characterizing the PCG, which is a directed cyclic graph Table 3 shows the parameters used in this characterization

To facilitate experimentation, we developed a D/KB generator that accepts parameter values as input and creates "random" data and rule bases satisfying the specified parameter values For rule bases, the D/KB generator accepts these parameters as input and creates a "random" directed cyclic graph satisfying these parameters It then generates a rule base, such that the PCG of this rule base is the above cyclic graph In general, there are several rule bases satisfying this condition, the D/KB generator generates one such

5 3 Tests and Results

This section describes the performance measurement and evaluation tests we performed using the testbed, the results of the tests, and an analysis of the results The tests are categorized into two groups (1) tests relating to D/KB query processing and (2) tests relating to D/KB updates

5 3 1 Tests Relating to D/KB Query Processing

The tests relating to D/KB query processing can be further categorized into two groups (a) tests relating to D/KB query compilation and (b) tests relating to D/KB query execution

5 3 1 1 Tests Relating to D/KB Query Compilation

After making several measurements, we found that the parameters that had the most effect on D/KB query compilation time were (i) the total number of stored rules and the number of rules relevant to the query, R_s and R_{sr}, which affect the time to extract the relevant rules from the Stored D/KB, and (ii) the number of stored and derived predicates relevant to the query, P_s and P_r, that affect the time to read the D/KB data dictionaries The purpose of the tests below is to study the effect of these parameters on D/KB query compilation time

Test 1 *Study the effect of the total number of rules in the Stored D/KB, R_s, and the number of Stored D/KB rules relevant to a query R_{sr}, on the time to extract the relevant rules from the Stored D/KB $t_{extract}$*

R_s was varied from 29 to 205 in steps of 16 and the value of $t_{extract}$ for a query with $R_{sr} = 2$ was recorded This procedure was repeated for queries with $R_{sr} = 7$ and $R_{sr} = 20$

Parameter		Description
D/KBMS architecture related parameters		Optimization strategy
		LFP evaluation strategy
		Rule storage structures
Workload related parameters	R_s	Total number of rules in the Stored D/KB
	R_w	Total number of rules in the Workspace D/KB
	P_s	Total number of derived predicates in the Stored D/KB
D/KB query and update related parameters	R_{sr}	Number of Stored D/KB rules relevant to the query
	P_r	Number of derived predicates relevant to the query
	D_{sr1}	Total number of tuples in all relevant base relations
	D_{sr2}	Number of relevant tuples in the relevant base relations
	T_w	Number of edges in the transitive closure of the PCG of the Workspace D/KB rules after extracting the relevant rules from the Stored D/KB

Table 1 Parameters

Relation type	Parameters	Remarks
List	Number of lists their average length	The number of tuples in a data base with n lists of average length l is approximately $n(l-1)$
Full binary tree	Number of trees their depth	The number of tuples in a data base with n trees each of depth d is $n(2^d - 2)$
Directed acyclic graph	Number of tuples fan out fan in path length	Fan out and fan in respectively refer to the number of arcs leaving and entering a node in the graph Path length refers to the number of nodes in a path starting with a node with zero fan in and ending with a node with zero fan out
Directed cyclic graph	Number of tuples fan out fan in path length number of cycles in the graph their average length	

Table 2 Extensional database characterization

Number of cliques
Average number of derived predicates in each clique
Average number of rules defining a derived predicate
Probability that a rule will be recursive
Fan out
Fan in
Path length
Number of cycles in each clique
Average cycle length

Table 3 Intensional database characterization

Figure 7 shows the results of this experiment For each query, notice that $t_{extract}$ is relatively insensitive to R_s This is explained by observing that of the two relations, *irulesource* and *ireachablepreds*, that are joined during the extraction of the relevant Stored D KB rules (see section 4 1), the former is relatively small, while even though the latter may be large, it has an index on its join column

Notice in Figure 7 that for a given value of R_s, $t_{extract}$ increases with R_{sr}, the number of rules in the Stored D/KB relevant to the query The curves of $t_{extract}$ versus R_{sr} in Figure 8 show this more clearly

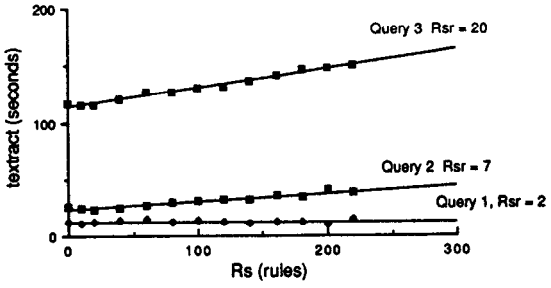

Figure 7 $t_{extract}$ versus R_s

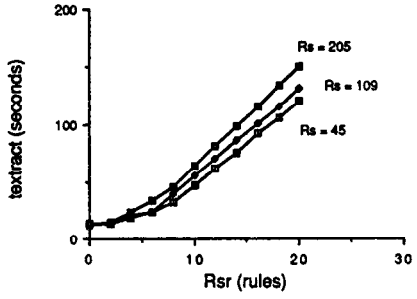

Figure 8 $t_{extract}$ versus R_{sr}

This reason for the behavior shown in Figure 8 is that R_{sr} is related to the join selectivity of the SQL query used to extract the relevant rules

The data in Figure 7 is for the case where the transitive closure of the PCG is physically stored in the Stored D/KB If rules are stored in raw source form only, or if the stored compiled form is just the PCG (as opposed to its transitive closure), the transitive closure of the PCG would have to be computed during query compilation We did not make quantitative measurements of $t_{extract}$ versus R_s for these cases, since we know from our previous work on algorithms for computing the transitive closure of a database relation that the performance deteriorates rapidly for very large relation sizes [14]

Test 2: *Study the effect of the total number of derived predicates in the Stored D/KB, P_s, and the number of derived predicates relevant to the query, P_r on the time to read the D/KB data dictionaries* The purpose of reading these dictionaries is to determine the types of the columns of the base and derived predicates prior to executing the type inferencing algorithm The motivation for doing this experiment is that reading the D/KB data dic tionaries involves accessing the Stored D/KB, which impacts performance The procedure here was basically the same as that in the preceding experi ment The values of P_r were 1, 4, and 10 for the three queries

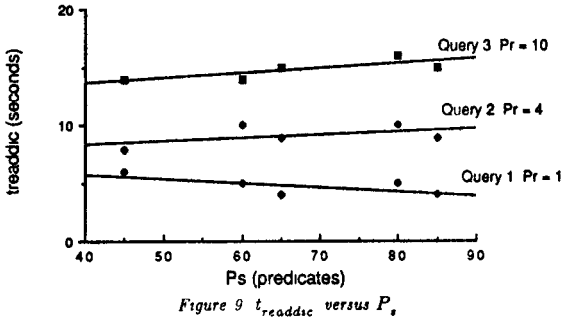

Figure 9 $t_{readdic}$ versus P_s

Figure 9 shows a plot of $t_{readdic}$ versus P_s Notice that for a given value of P_r, $t_{readdic}$ is relatively insensitive to P_s To see why this is so, let us look at how the intensional data dictionary is accessed for a query having say p_1 and p_2 as the relevant derived predicates This is accomplished via the following SQL query

```
SELECT *
FROM isystables, isyscolumns
WHERE isystables tabid = isyscolumns tabid AND
      (isystables tabname = p1 OR
       isy tables tabname = p2))
```

The execution time of the above query is insensitive to P_s (the number of tuples in *isystables*), because we place indexes on *isystables* and *isyscolumns*

Also notice that for a given value of P_s, $t_{readdic}$ increases with P_r This is because P_r is related to the join selectivity of the above query

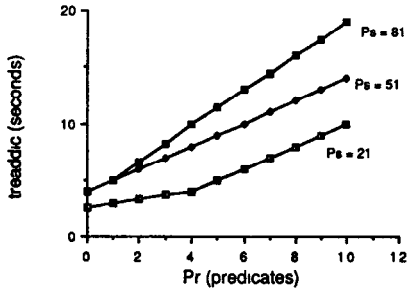

Figure 10 $t_{readdic}$ versus P_r

Figure 10 shows a plot of $t_{readdic}$ versus P_r for three different values of P_s We haven t shown a plot of $t_{readdic}$ versus R_s, but we found $t_{readdic}$ to be insensitive to R_s This is because R_s affects $t_{readdic}$ in so far as it affects P_s, the number of derived predicates in the Stored D/KB, and as we explained above, $t_{readdic}$ is insensitive to P_s

Test 3 *Study the relative contributions of the different steps in D/KB query compilation on the total compilation time t_c* After making several measurements of the different steps in D/KB query compilation, we found that the principal contributions to t_c came from the following five components

- t_{setup} time to set-up the query related data structures,
- $t_{extract}$ time to extract relevant rules from the stored D/KB,
- $t_{readdic}$ time to read the D/KB data dictionary,
- t_{order} time to generate the evaluation order list, and
- $t_{compile}$ time to compile the program segment and link it with the run-time library

R_{sr}	t_{c2}	t_{c4}	t_{c6}	t_{c8}	t_{c11}
1	6%	25%	8%	13%	48%
7	5%	38%	10%	11%	36%
20	4%	67%	8%	6%	15%

Table 4 Breakup of D/KB query compilation time

Table 4 shows the contributions of these components for three different queries, with R_{sr} equal to 1, 7, and 20 Notice that as R_{sr} increases from 1 to 20, the relative contribution of $t_{extract}$, the time to extract the relevant Stored D/KB rules, increases from 25% to 67% Also, the rate of increase appears to be quite rapid

t_{order} appears to be making a non-trivial contribution to t_c This is actually due to the fact that in our testbed, the evaluation order list is computed by making a Unix system call to execute the Unix topological sort utility The overhead imposed by the system call is particularly significant We could have avoided making the system call by implementing a topological sort algorithm, this would have had the effect of making the contribution of t_{order} insignificant

Finally, note that the relative contribution of $t_{compile}$, the time to compile the code fragment generated by the Knowledge Manager and link it with the run time library appears quite significant However, this is very much compiler dependent and can vary greatly from system to sys tem We can make a similar observation about t_{setup}

5 3 1 2 Tests Relating to D/KB Query Execution

Quantitative analysis of D/KB query execution performance is complicated by the fact that the execution time is greatly influenced by the nature of the query and data This is because the type of query and data greatly affect the size of the set of relevant facts, D_{sr2}, and the amount of duplicate work done during LFP computation, which were shown in [15] to be two of the most important parameters influencing D/KB query execution time The principal purpose of the tests described in this section is to study the effect of these parameters on D/KB query execution time

The tests all use the *ancestor* query

$$ancestor(X, Y) \leftarrow parent(X, Y)$$

$$ancestor(X, Y) \leftarrow parent(X, Z), ancestor(Z, Y)$$

$$query(X) \leftarrow ancestor(\ john\ , X)$$

and tree structured data for the *parent* relation The results will obviously be different for other queries and data types Still, we can draw some general conclusions from our test results

When studying the effect of redundant work, we didn't directly measure this parameter Rather, we measured the execution times for naive and semi-naive LFP evaluation, the difference between the two indicating the impact of redundant work

We now describe the tests in more detail

Test 4 *Study the effect of the fraction of relevant facts, D_{sr2}/D_{sr1}, on D/KB query execution time, t_e* We varied D_{sr2}/D_{sr1} in two different ways In the first method, we kept D_{sr1} fixed by keeping the size of the *parent* relation fixed and varied D_{sr2} by rooting the *ancestor* query at different subtrees of the *parent* relation Thus, each value of D_{sr2} was obtained from a different query, each query having a different constant In the second method, we kept D_{sr2} fixed by fixing the query constant and varied the size of D_{sr1} by making the *parent* relation progressively larger Here, the same query was applied to *parent* relations of different sizes Semi-naive evaluation was used for LFP computation Optimization was not used

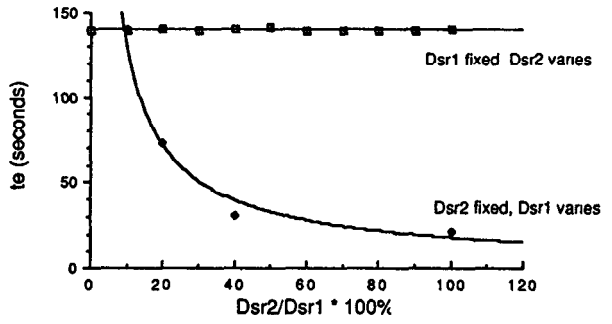

Figure 11 t_e versus D_{sr2}/D_{sr1}

Figure 11 shows a plot of t_e versus D_{sr2}/D_{sr1} Notice that when D_{sr1} is fixed, t_e is insensitive to D_{sr2} This is because in the absence of the magic set optimization, the transitive closure of the entire *parent* relation is computed, regardless of the percentage of this relation that is actually relevant to the query We study the impact of this optimization in another experiment

On the other hand, when D_{sr2} is fixed but D_{sr1} is not, t_e increases with D_{sr1} (equivalently, t_e decreases with D_{sr2}/D_{sr1}) This is because the transitive closure is being computed for progressively larger relation sizes

Test 5 *Study the impact of redundant work done during LFP computation on D/KB query execution time* We first measured t_e for several *ancestor* queries rooted at different sub-trees of the *parent* relation, keeping D_{sr1} fixed and using semi-naive LFP evaluation We then repeated this procedure for naive LFP evaluation

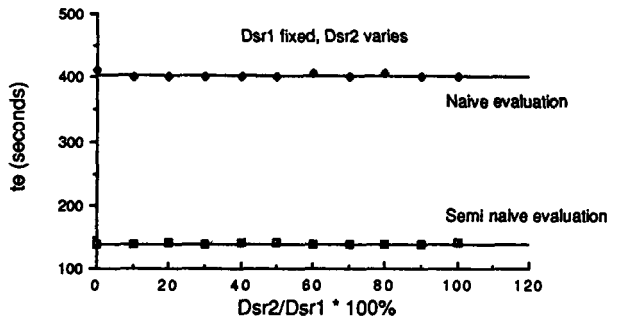

Figure 12 Comparison of two LFP evaluation methods

Figure 12 shows a plot of t_e versus D_{sr2}/D_{sr1} for both naive and semi-naive evaluation Notice that for the query and database in this test, semi-naive evaluation is between 2 5 to 3 times faster than naive evaluation The difference is due to the fact that semi-naive evaluation avoids a lot of duplicate work by computing only the differential of $f(R)$ during each iteration when evaluating the LFP of $R = f(R)$ Naive evaluation, on the other hand, recomputes tuples computed in previous iterations

Test 6 *Study the relative contributions of the various steps in naive and semi naive LFP evaluation.* There are three major steps when naive and semi-naive LFP evaluation are implemented as an application program and a relational DBMS create and clean up temporary tables, evaluate right hand side of recursive equations, and check termination conditions

Table 5 shows the results of this test Notice that in naive evaluation, 95% of the time is spent in evaluating the right hand side of the recursive equations and doing the termination check, while the corresponding activities consume 82% of the time in semi-naive evaluation The activities are not quite the same since semi-naive evaluation computes only the differential of the right hand side of the recursive equations during each iteration

Table 5 also shows a comparison of the actual time taken to evaluate the right hand side of the recursive equations (or the differential in semi-naive evaluation) and to do the termination check for naive and semi-naive evaluation Notice that the times for naive evaluation are about 2 5 to 3 times greater than those for semi-naive evaluation This is

Subtask	Naive		Semi Naive	
	time	percentage	time	percentage
Create and clean up temporary tables	20	5%	24	18%
Evaluate right hand side of recursive equations	110	28%	42	31%
Check termination conditions	260	67%	69	51%
Total	390	100%	135	100%

Table 5 Breakup of LFP evaluation time

the principal reason semi-naive evaluation was found to be 2 5 to 3 times faster than naive evaluation in the previous test

The termination check is expensive in our implementation because the SQL interface between the Knowledge Manager and the DBMS forces a set difference, an expensive operation, to be computed during this check

Test 7 *Study the impact of optimization on D/KB query execution time* The generalized magic set algorithm was used as the optimization method In this algorithm, the original query and rule set are rewritten as two sets of rules magic rules and modified rules The execution time t_e is the sum of evaluate these rule sets

This test comprises three parts First, we measured t_e as a function of the *query selectivity*, D_{sr2}/D_{sr1} (the ratio of the relevant tuples and the total number of tuples) for the four cases resulting from using naive and semi-naive evaluation with and without optimization D_{sr2}/D_{sr1} was varied by keeping D_{sr1} fixed and varying D_{sr2} Figure 13 shows the results of this test

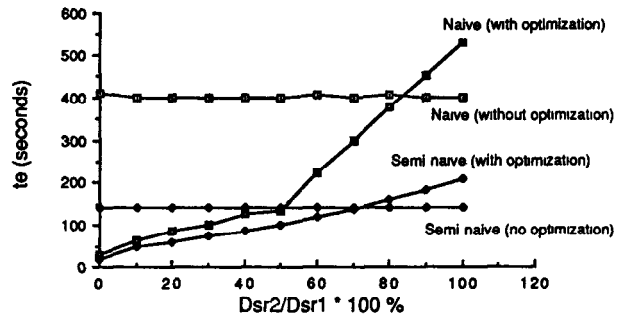

Figure 13 Effect of optimization

Notice the significant impact of optimization for queries with low selectivity (Figure 13) For example, when only 5% of the base relation tuples are relevant, the execution time with optimization is about 6 to 7 times faster than without optimization The impact of optimization is particularly significant for very large base relations and very low query selectivity In the second part of this test, we executed an *ancestor* query with very low selectivity (05%) against a *parent* relation containing over 16,000 tuples and measured t_e with and without optimization We found that without optimization, the query took several orders of magnitude longer to execute than it did with optimization! We expect that in very large database environments, the query selectivity will be small in many cases and so, this test represents a very plausible scenario

Notice that t_e is insensitive to the query selectivity in the absence of optimization, since the transitive closure of the entire *parent* relation is computed in this case However, with optimization t_e increases with the query selectivity since in this case the transitive closure is computed only for the relevant portion of the *parent* relation, which grows progressively larger when D_{sr2}/D_{sr1} increases

In fact, there is a crossover point beyond which optimization results in higher query execution times This typically happens when the selectivity of the query is high, i e , when most of the extensional database is relevant To understand why this is so recall that in the magic set strategy, an LFP computation is done is first to evaluate the magic rules and determine the set of relevant facts Then, another LFP computation is done to evaluate the modified rules and determine the query results against the magic set, i e the result relations from the first LFP evaluation When the query selectivity is low, the size of these relations is small enough so that the two LFP computations together take less time than a single LFP computation on the original base relations However, when the selectivity of the query is high, the size of the magic set predicates is large and the extra overhead in first computing them results in a higher overall execution time

The crossover selectivity where optimization degrades performance for semi-naive evaluation is about 72%, while it is about 82% for naive evaluation The higher crossover point for naive evaluation is due to the fact that optimization has a bigger impact on naive evaluation as it does a lot of redundant work

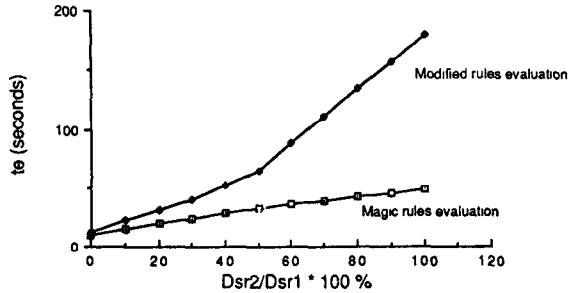

Figure 14 Magic and modified rule evaluation time

Figure 14 is a plot showing the execution times for the two LFP computations as a function of D_{sr2}/D_{sr1}. The rate of increase of the magic rules evaluation is lower than that for the modified rules evaluation. This is because the magic rules evaluation time depends mostly on D_{sr1}, the size of the base relations, which was fixed in this test. On the other hand, the modified rules evaluation time is quite sensitive to D_{sr2}, the number of relevant facts, which was varied.

5.3.2 Tests Relating to D/KB Updates

This section describes tests to study the effect of R_s, R_w, and T_w on D/KB update time, t_u.

Test 8 *Study the effect of R_s on the D/KB update time, t_u.* We loaded the Workspace D/KB with a single rule and updated the Stored D/KB, varying the value of R_s from 9 to 189.

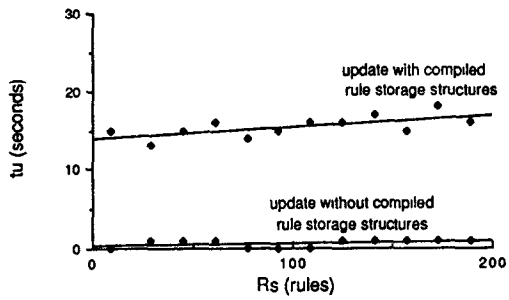

Figure 15 t_u versus R_s

Figure 15 shows a plot of t_u versus R_s both with and without compiled rule storage structures. In the latter case, the update time is simply the time to store the source form of the rules. Notice that updates are almost an order of magnitude faster without compiled form rule storage.

Also, t_u is relatively insensitive to R_s. The main reason for this is that the time to extract the relevant rules from the Stored D/KB is a significant contributor to t_u (see next experiment). This insensitivity of t_u to R_s is significant because it means that the D/KB update time does not degrade for very large rule sets.

Test 9 *Study the relative contributions of the different components of t_u as a function of R_w and T_w.* The query update time t_u consists of three major components:

- t_{u1} Time to extract the rules relevant to the Workspace D/KB from the Stored D/KB
- t_{u2} Time to update the *irulesource* relation
- t_{u3} Time to update the *ireachablepreds*, *isystables*, and *isyscolumns* relations

We updated a Stored D/KB with R_s = 189 with a Workspace D/KB containing 38 rules and measured the values of t_{u1}, t_{u2} and t_{u3}. The value of T_w was 137, i.e., the transitive closure of the PCG of the Workspace D/KB rules after extracting the relevant rules contained 135 edges. We then repeated this procedure for a Workspace D/KB containing 1 rule and T_w = 21. Table 6 shows the results.

T_w	R_w	R_s	t_{u1}	t_{u2}	t_{u3}
137	38	189	42%	13%	45%
21	1	189	81%	6%	13%

Table 6 Breakup of D/KB update time

The main point to note is that t_{u1}, the time to extract the relevant rules is a significant component of t_u. For small values of R_w and T_w, this time in fact makes up the bulk of the contribution to t_u. However, for large values of R_w and T_w, the percentage contribution of t_{u1} decreases, since that of t_{u3} increases. Also, notice that the time to store the source form of the rules, t_{u2}, contributes only a small amount to the overall D/KB update time.

6 Conclusions

We have presented the design and implementation of a data/knowledgement management testbed, including the system architecture and major data structures and algorithms. We also presented and analyzed the results of several experiments we performed to quantitatively measure D/KBMS performance and to study performance as a function of various parameters. In this section, we draw some conclusions based on the test results and our experience.

1. Rule storage structure design is a major design issue in D/KBMS design. In our testbed, in addition to the source form storage, the rule storage structures include the transitive closure of the PCG of the rules (all reachable predicates from a predicate). We believe that, in order for a D/KBMS to be scalable to handle large rule sets, it is important that the rule storage structures be such that the time to extract the relevant rules is independent of the total number of rules in the Stored D/KB. Otherwise, the D/KB query compilation times will grow with the size of the rule base. Our experimentation has shown that the transitive closure of the PCG with appropriate indexes achieves the effect of making the relevant rules extraction time independent of the rule base size. This is because with this storage structure, the time to extract the relevant rules depends only on the number of rules extracted and not on the total number of rules.

   There are two important tradeoffs that relate to rule storage structures. The first is a time-vs-space tradeoff. Compiled form rule storage structures like the transitive closure of the PCG use more space but permit faster query compilation than non-compiled storage structures. The other tradeoff is between query compilation time and update time. Compiled form storage structures take longer to update, sometimes even an order of magnitude longer as some of our experiments indicated, than non-compiled storage structures. The choice of rule storage structure must be dictated by the relative cost of storage versus compilation and by the application characteristics – whether it is query intensive or update intensive. For example, the PCG itself (as opposed to its transitive closure) has been proposed as a rule storage structure. However, this is not a good choice for query intensive applications. This is because during query compilation, the transitive closure of the PCG will have to be computed to extract the relevant rules and this can be very time consuming for rules with large PCGs.

2. As we argued before, from the D/KB query compilation point of view, a good rule storage structure is one where the relevant rule extraction time depends only on the number of rules extracted and not on the total number of rules. However, we found that the time to extract the relevant rules is *very* sensitive to the number of rules extracted. A key to avoiding excessive compilation times is to structure the rules in such a way that the number of relevant rules for a query is small. Object oriented database techniques can prove to very useful here. For example, a small set of rules can be encapsulated within an object and these rules can be retrieved whenever the object receives a message representing a query against them. Encapsulating rules within an object is a way of structuring the rules so that only the relevant portions of the rule base are processed during compilation. Of course, much work needs to be done to integrate the concepts of object-oriented database systems – inheritance, message processing, persistent objects, etc – with those of D/KB query and update processing.

3. Precompilation of D/KB queries can prove to be very useful. This is especially true for frequently occurring queries with large R_{sr} values. The price of precompilation is that, for precompiled queries, information about referenced relations and rules must be recorded. During updates, this information is checked to see whether the update invalidates any compiled query. However, for applications involving few updates and frequently occurring queries with large R_{sr} values, this price is well worth paying.

4. Two of the main parameters affecting D/KB query execution time are the ratio of relevant facts to total number of facts (D_{sr2}/D_{sr1}) and the amount of redundant work done in the LFP evaluation. To reduce the amount of redundant work and to restrict the LFP evaluation to the relevant database tuples, the D/KBMS architecture can use better evaluation methods such as semi-naive LFP evaluation and optimization strategies such as the generalized magic sets strategy.

   However, there is a tradeoff in using optimization strategies while optimization restricts LFP evaluation to the relevant tuples of the database, work must be done to first determine these tuples. There is a crossover value of the query selectivity beyond which optimization actually results in higher query execution times. Optimization pays best when this selectivity is low, i.e., for queries that retrieve only a small fraction of the database. The benefit of optimization is particularly telling for queries with very low selectivity and very large base relations. For such applications, we found that without optimization, the query took several orders of magnitude longer to execute than it did with optimization! We expect that in very large database environments, the query selectivity will be small in many cases and the use of optimization is highly recommended despite the extra work introduced. Ideally, the D/KBMS query optimizer should adapt the optimization strategy dynamically, switching it on for queries with low selectivity and off otherwise.

5. Relational algebra alone is not a good choice for the LFP evaluation since the evaluation has to be done via an application program and this introduces several inefficiencies. For example during each iteration of the while loop, several table copies are performed. Also, the

termination check becomes a very expensive operation since with relational algebra as the DBMS interface this involves computing a set difference In fact, with relational algebra, the "real" work in LFP evaluation, viz evaluating the right hand side of the recursive equations (or their differential), takes up only about 30% of the while loop execution time in our example query The rest of the time is spent doing table copies, termination checking, and clearing temporary tables Of course the time spent doing real work will increase for queries more complex than transitive closure Still, the overhead mentioned above will constitute a significant portion of the execution time

6 The above inefficiencies cannot be overcome using parallelism alone While a parallel relational database machine can certainly speed up table copying and termination checking, it does not significantly reduce the percentage contribution of these operations to the while loop execution time

7 To achieve high performance in D/KB query execution, it is very important that the relational algebra interface be augmented with a generalized LFP operator This operator should accept a set of recursive equations of the form, $r_i = f_i(r_1, \ldots, r_n)$, $i = 1, \ldots, n$, as input and compute their least fixed point, thereby solving each r_i By including such an operator in the DBMS interface, many of the inefficiencies that arise with relational algebra can be alleviated We mention several optimization possibilities that open up if the DBMS interface included an LFP operator, none of which are possible if the interface was just relational algebra

   a Table copying can be avoided by manipulating buffer pointers

   b The full set difference operation during termination checking can be avoided This is because as soon as a tuple is found that was not computed in the previous iteration, the termination check can stop

   c A dynamically adaptable indexing strategy can be designed to speed up the evaluation of the right hand side of the recursive equations or their differential This strategy would dynamically create and drop temporary indexes on the base and intermediate derived relations depending on their relative sizes A new indexing technique, proposed by Lu and Shan [16] is one effort along this direction

   d The join strategy can be dynamically changed between iterations if necessary, depending on the sizes of the base and intermediate derived relations and the join selectivities from the previous iterations

8 The performance of LFP evaluation can be significantly improved by parallel and pipelined processing We list several possible strategies below

   a During each iteration, the right hand side of each recursive equation may be evaluated in parallel

   b Pipelining and data flow techniques may be used to evaluate the relational algebra tree corresponding to the right hand side of these equations

   c Parallel join algorithms may be employed during this evaluation

9 In addition to a general LFP operator, the DBMS interface should include commonly occurring special LFP operators, such as transitive closure and the alpha operator [17] This is because it may be possible to optimize the execution of such special operators better than that of a general LFP operator In general, it will be difficult for the query optimizer to recognize that a given set of LFP equations corresponds to one or another specialized LFP operator Therefore, the Knowledge Manager interface should include ways of denoting such operators Then the Knowledge Manager can generate code containing them and the DBMS can execute this code efficiently

Finally, we would like to point that the current version of the testbed is far from a complete D/KBMS In addition to the unimplemented components and algorithms (such as the magic sets optimization strategy), there are a large number issues related to D/KB query processing that are not addressed at all These include complex terms in Horn clauses [18], extension of Horn clauses to include negation, the safety check for recursive queries [19], and the consistency check and truth maintenance of the knowledge base However, as we can see from the experiments described in the paper, this testbed is a valuable experimental tool for further research in D/KBMS design

Acknowledgement

The authors would like to thank the other members of the project, especially James Richardson, with whom we had many helpful discussions, Amit Sheth, and Allan Anderson

References

1 L Daval *et al*, PROBE Technical Report, CCA 85 03,, Computer Corporation of America,, July 1985

2 K Morris, J Ullman and A van Gelder, "Design Overview of the NAIL! System, ' Proceedings of the 3rd International Conference on Logic Programming, London, England, July 1986

3 'Database Research at MCC, ' Proceedings of the the 12th International Conference on Very Large Data Bases, Kyoto, Japan, August 1986

4 D P McKay and S C Shapiro, ' Using Active Connection Graphs for Reasoning with Recursive Rules, Proceedings 7th International Conf on Artificial Intelligence, pp 368-374, August 1981

5 L J Henschen and S A Naqvi, "On Compiling Recursive Queries in First-order Databases," *JACM*, vol 31, no 1, pp 47 85, Jan 1984

6 F Bancilhon, "Naive Evaluation of Recursively Defined Relations in *On Knowledge Base Management Systems – Integrating Database and AI Systems*, ed Brodie and Mylopoulos, Springer-Verlag, 1985

7 F Bancilhon *et al*, "Magic Sets and Other Strange Ways to Implement Logic Programs," Proc 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986

8 D Sacca and C Zaniolo, "On the Implementation of a Simple Class of Logic Queries for Databases," 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems, 1986

9 D Sacca and C Zaniolo, "The Generalized Counting Method for Recursive Logic Queries," Proc ICDT, 1986

10 C Beeri and R. Ramakrishnan, "On the Power of Magic," Proc of 6th ACM SIGMOD-SIGACT symposium on Principles of Database Systems, May, 1987

11 A V Aho and J D Ullman, "Universality of Data Retrieval Languages," Proc Sixth ACM Symp on Principles of Programming Languages, pp 110-117, Jan 1979

12 I Balbin and K Ramamohanarao, "A Differential Approach to Query Optimization in Recursive Deductive Databases. Technical Report 86/7, Department of Computer Science, University of Melbourne, Australia, 1986

13 R Ramnarayan, et al, *Very Large Parallel Data Flow Final Report* Rome Air Development Center, Contract No F30602-85-C-0215 December 1987

14 H Lu, K Mikkilineni, and J Richardson, 'Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation," Proc 3rd Internationsl Conf on Data Engineering, Los Angeles, CA, February 3-5 1987

15 F Bancilhon and R Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," Proc SIGMOD, Invited Paper, 1986

16 H Lu and M C Shan, *B* -Tree A New Access Method Supporting the Least Fixpoint Computation for Recursive Relations, November 1987 submitted for publication

17 R Agrawal, "Alpha An Extension of Relational Algebra to Express a Class of Recursive Queries," Third International Conference on Data Engineering, pp 580-590, IEEE, Los Angeles, February 3-5, 1987

18 C Zaniolo, "The Representation and Deductive Retrieval of Complex Objects, ' Proceedings of 11th International Conference on Very Large Data Bases, Stockholm, Sweden, September 1985

19 C Zaniolo, 'Safety and Compilation of Non-Recursive Horn Clauses," Proceedings of the 1st International Conference on Expert Database Systems, Charleston, April 1986