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 were conducted using the current version of the testbed and the conclusions from the test results are presented. The principal contributions of our work in designing and implementing the testbed have proven very useful for further investigation of technical issues in D/KBMS design.

1 Background Concepts in D/KB Query Processing

1.1 Data/Knowledge Base

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

\[
\text{head - body}
\]

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.

\[
\begin{align*}
R_1 & : p(X, Y) \leftarrow p(X, Z), q(Z, Y) \\
R_2 & : p(X, Y) \leftarrow b_1(Y) \\
R_3 & : p(X, Y) \leftarrow b_2(Z), p(Z, Y) \\
R_4 & : p(X, Y) \leftarrow b_2(Y) \\
R_5 & : p(X, Y) \leftarrow b_1(Z), p(Z, Y) \\
R_6 & : q(X, Y) \leftarrow p(X, Y), p(Z, Y)
\end{align*}
\]

Figure 1 Sample data/knowledge base

The logical interpretation of a Horn clause is that the body implies the head. For example, \( R_1 \) in Figure 1 means that for all \( X, Y, Z \), if \( p(X, Z) \) and \( q(Z, Y) \), then \( p(X, Y) \). 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 rules. 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 \( p \), \( p_1 \), 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.

\[
\begin{align*}
\text{Set}_1 & : p(X, Y) \leftarrow \text{e}(X, Z), b(Z, Y) \\
\text{Set}_2 & : p(a, e) \\
\end{align*}
\]

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 motivation for distinguishing between the intensional and extensional knowledge bases is that relations defining base predicates are stored into D/KBS design gleaned from the test results and our design and implementation experience.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.


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/KBMAs.

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 z as its head and z is reachable from p.

For example, in Figure 1, p1 is reachable from p so is b1, because b1 is reachable from p1 and p2 is reachable from p1. p2 and p3 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 - q1, q2, ... qn is called a recursive rule if at least one of the qi's is mutually recursive to p. Rules are mutually recursive if the predicates in their heads are mutually recursive. Rules that do not recur are not to be 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 an formalism. Each node in a PCG represents a predicate. Edges arise from rules if there is a rule of the form p = q1, q2, ... qn. There is a directed edge from each to p of the qi'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 clique. Here, by clique we 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.

Figure 3: Cliques for rules in Figure 1

Clique Recursive predicates Recursive rules Exit rules
1 p1, q1 R1, R2 R3
2 p1 R1, R2 R3
3 p2 R3, R5 R6

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 - C indicates that some predicate in the clique C appears in the body of a rule defining P.
(2) C - P indicates that P appears in the body of a rule defining some predicate of C.
(3) P1 - P2 indicates that P1 appears in the body of a rule defining some predicate of P2.
(4) C1 - C2 indicates that some predicate of C1 appears in the body of a rule defining some predicate of C2.

While the PCG may be a cyclic graph, the evaluation graph can be 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 if some of these predecessors define 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.

Locally, 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 (C2, C3, C1, query) and (C3, C2, C1, 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 to produce the head predicates till the predicate query is generated. Top-down evaluation starts with the query and keeps evaluating predicates in the body of the relevant rules by propagating the bindings in the head predicates of these rules.

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 = f(r1, r2, ..., rn), 1 = 1, ..., n, where the f's are relational algebra expressions.

Examples of top-down evaluation are the Hennessen-Nagv 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 (e.g., 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 additional information passing (up) to propagate the information in the query (query constants) and restrict the computation to tuples that are related to the query. Been and Ramakrishna 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 optimisation 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 Tested Architecture

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

- Knowledge Manager (KM), which compiles intensional database systems using appropriate compilation tools into one 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. DBMSs 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 system called the Knowledge Manager. 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 platform and an evaluation platform. As a demonstration platform, the testbed illustrates the motivation and basic functionality of a DKBMS, the components of a DKBMS architecture, alternative implementations of user components and their relative tradeoffs, and the factors contributing to DKB query compilation and execution time. As an 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 starts by creating a set of rules and facts. These rules and facts are stored in a memory resident environment called the Workspace DKB The Workspace DKB 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 DKB After entering a set of rules and facts into the Workspace DKB, the user issues queries against them. If he is satisfied that the rules and facts in the workspace DKB are correct, he updates the stored DKB with these rules and facts.

The overall configuration of the testbed is shown in Figure 5.

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 DKB 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 acquires the data/knowledge base and returns the query results. The Knowledge Manager also handles updates to the Stored DKB. The DKB is a commercial relational database management system with SQL and embedded SQL (in C) interfaces.

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 DKB Manager, Stored DKB Manager, Semantic Checker, Optimizer, and Code Generator. The Knowledge Manager architecture is shown in Figure 6. The circles in this figure represent data structures and the boxes, components.

3.2.1 Rule Parser

The Rule Parser parses the input Horn Clauses. It accepts Horn clauses and the rules and facts from the user interface and compiles queries into code fragments. It consists of the following components: Rule Parser, Workspace DKB Manager, Stored DKB Manager, Semantic Checker, Optimizer, and Code Generator. The Knowledge Manager architecture is shown in Figure 6. The circles in this figure represent data structures and the boxes, components.

3.2.2 Workspace DKB Manager

The Workspace DKB Manager provides the following functions:
- Determine all predicates reachable from a given predicate
- Find the cliques in the workspace DKB
- Generate the evaluation order list

3.2.3 Stored DKB Manager

The Stored DKB Manager provides the following functions:
- Extract rules from the stored DKB 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 DKB with rules and facts from the Workspace DKB

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 user, 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

The DKB query processing performance depends upon the efficiency of LFP computation. The Optimizer uses the generalized magic sets algorithm proposed by Beenz 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. As we mentioned before, SQL is not the only KB interface to use when designing a high performance D/KBMS, since relational algebra cannot express LFP queries. 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 D/KBMS to give the query results.

3.4 Control Flow

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

The User Interface accepts user query and passes it to the D/KB Manager, which in turn determines all predicates reachable from the query. In general, there may be rules defining these predicates in the Stored D/KB. The 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 D/KB Manager analyzes each query, 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 generalization magic rules strategy, it generates three sets of rules in the workspace: adored, magic, and modified rules. It also generates an adored version of the query.

The workspace manager finds the clauses in the obtained rules and uses this information to generate the evaluation order list.

The Semantic Checker now performs the two tasks:
- 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 tester can be grouped under four categories:

(i) Algorithms that perform specific tasks on data structures, 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 algorithms and the algorithm for set ordering the relevant rules from the Stored D/KB during compilation and updates.

(iii) Algorithms that unify various 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 efficiently generating information passing (IP) strategy as part of the generalized magic set optimization technique and Ramakrishnan in their paper [10].

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

4.1 Intensional Database Storage Structures

The D/KB stores both the intensional and intensional database storage structures. The 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 tester is a commercial relational database system, the intensional database is stored as a set of relations. This set consists of relations, such as attributes, tuples, and rules. The first two relations, attributes and tuples, 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:

<table>
<thead>
<tr>
<th>Relation Name</th>
<th>Column Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>char, integer</td>
</tr>
<tr>
<td>tuples</td>
<td>tabled integer, column char, number integer, column char</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Relation Name</th>
<th>Column Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>rulesource</td>
<td>headprename char, topredname char</td>
</tr>
</tbody>
</table>

treachablepreds 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 rulesource and constitutes the compiled form of these rules. It has the following schema:

<table>
<thead>
<tr>
<th>Relation Name</th>
<th>Column Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>reachablepreds</td>
<td>frompredname char, topredname char</td>
</tr>
</tbody>
</table>

We use reachablepreds 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:

\[
q(X, Y) = \mu(X, Z) . \varphi(Z, Y)
\]

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

```
SELECT rulesource rule
FROM rulesource, reachablepreds
WHERE reachablepreds frompredname = rulesource headprename
OR reachablepreds frompredname = rulesource headprename
AND ( reachablepreds topredname = p
OR reachablepreds topredname = 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 rulesource and reachablepreds are indexed.

4.2 D/KB Query Processing Algorithm

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

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 rules reachable from the predicates in P and the rules whose head predicates are in P, using a query similar to the SQL query shown in section 4.1, insert them into P and R, respectively.

   1.4 Load 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 P and R.

2. Construct the PCG for the rule set R and find the clauses 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 clauses and non-recursive predicates as per the total order prescribed by the evaluation order list. If the node to be evaluated is a clause, optionally apply an optimization strategy to the clause 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 6). 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 tested 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 difference between the existing version of the Stored D/KB and the rules stored in the Workspace D/KB. This can be done using a query similar to the SQL query shown in section 4.1. Let DKB\_\text{new}\ denote the extracted rules.

2. Construct the PCG of the rules in ΔDKB\_\text{new}\. ΔDKB = ΔDKB \setminus DKB\_\text{new}\, which denotes the set of rules which are either in ΔDKB or in DKB\_\text{new}\.

3. Compute the transitive closure of this PCG. This gives all the predicates reachable from a given predicate in ΔDKB\_\text{composite}.

4. Perform the type checking algorithm which is described next in this section.

5. For each derived predicate p in ΔDKB\_\text{composite}, add tuples to the stored rules if information on p is not present in these tables.
For each derived predicate \( p \) in \( D/KB \), we add tuples to \( \text{treachablePreds} \) by looking at the transitive closure computed in step \( 3 \).

For each rule in \( D/KB \), we add tuples to \( \text{trueSource} \).

We mentioned before that \( \text{treachablePreds} \) is the transitive closure of the PCC of the rules in \( \text{trueSource} \). 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 (\( \text{IDB}_\text{new} \)), and not on the entire stored \( D/KB \). This can result in substantial savings in update times for very large rule sets as the size of \( \text{IDB}_\text{new} \) 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 \( D/KB \) performance and to understand \( D/KB \) performance sensitivity and behavior with respect to various system parameters. The main 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, \( t_f \), (ii) \( D/KB \) query execution time, \( t_e \), and (iii) \( D/KB \) update time, \( t_u \). These measures are composed 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 \( D/KB \) use, 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 experimentations: lists, full binary trees, directed acyclic graphs, and directed cyclic graphs. 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 \( D/KB \) use, and the query and update parameters relate to the portions of the \( D/KB \) relevant to the query. Table 1 describes these parameters.

Table 1 Parameters

<table>
<thead>
<tr>
<th>Relation type</th>
<th>Parameters</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>Number of lists their average length</td>
<td>The number of tuples in a data base with ( n ) lists of average length ( l ) is approximately ( n(l - 1) )</td>
</tr>
<tr>
<td>Full binary tree</td>
<td>Number of trees their average depth</td>
<td>The number of tuples in a data base with ( n ) trees each of depth ( d ) is ( n(2^d - 2) )</td>
</tr>
<tr>
<td>Directed acyclic graph</td>
<td>Number of tuples fan out fan in path length</td>
<td>Fan out and fan in respectively refer to the number of edge 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</td>
</tr>
<tr>
<td>Directed cyclic graph</td>
<td>Number of tuples fan out fan in path length</td>
<td>Number of cycles in the graph whose average length</td>
</tr>
</tbody>
</table>

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: (a) tests relating to \( D/KB \) query processing and (b) 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.

5.3.2 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, \( R_r \) and \( R_s \), 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, \( P_{sr} \) and \( P_{dr} \), 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.

Table 2

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_r )</td>
<td>Total number of rules in the Stored ( D/KB )</td>
<td>Probability that a rule will be found</td>
</tr>
<tr>
<td>( R_s )</td>
<td>Total number of rules in the Workspace ( D/KB )</td>
<td>Path length</td>
</tr>
<tr>
<td>( P_{sr} )</td>
<td>Total number of derived predicates in the Stored ( D/KB )</td>
<td>Number of cycles in a path</td>
</tr>
<tr>
<td>( P_{dr} )</td>
<td>Number of derived predicates relevant to the query</td>
<td>Average cycle length</td>
</tr>
</tbody>
</table>

5.3.3 Tests Relating to \( D/KB \) Query Execution

After making several measurements, we found that the parameters that had the most effect on \( D/KB \) query execution time were: (i) the total number of stored rules and the number of rules relevant to the query, \( R_r \) and \( R_s \), 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, \( P_{sr} \) and \( P_{dr} \), 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 execution time.

Table 3

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R_r )</td>
<td>Total number of rules in the Stored ( D/KB )</td>
<td>Probability that a rule will be found</td>
</tr>
<tr>
<td>( R_s )</td>
<td>Total number of rules in the Workspace ( D/KB )</td>
<td>Path length</td>
</tr>
<tr>
<td>( P_{sr} )</td>
<td>Total number of derived predicates in the Stored ( D/KB )</td>
<td>Number of cycles in a path</td>
</tr>
<tr>
<td>( P_{dr} )</td>
<td>Number of derived predicates relevant to the query</td>
<td>Average cycle length</td>
</tr>
</tbody>
</table>
Figure 7 shows the results of this experiment. For each query, notice that \( t_{extract} \) is relatively insensitive to \( R \). This is explained by observing that of the two relations, \( R \) and \( \text{table} \), the latter is relatively small, while even though the former may be large, it has an index on its join column.

Notice in Figure 7 that for a given value of \( R \), \( t_{extract} \) increases with \( R \), the number of rules in the Stored D/KB relevant to the query. The curves of \( t_{extract} \) versus \( R \) in Figure 8 show this more clearly.

The reason for the behavior shown in Figure 8 is that \( R \), 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 PCC is physically stored in a relation in the Stored D/KB. If rules are stored in a relation, then of the stored compiled form of the PCC (as opposed to its transitive closure), the transitive closure of the PCC would have to be computed during query compilation. We did not perform quantitative measurements of \( t_{extract} \) versus \( R \) 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, \( P \), and the number of derived predicates relevant to the query, \( P_r \), on the time to read the D/KB data dictionary. 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 inference algorithm. The motivation for doing this experiment is that reading the D/KB data dictionary is a frequent operation that is significant in terms of its impact on performance. The procedure here was basically the same as that done in the preceding experiment. The values of \( P \) were 1, 4, and 10 for the three queries.

Figure 9 shows a plot of \( t_{read} \) versus \( P_r \). Notice that for a given value of \( P \), \( t_{read} \) is relatively insensitive to \( P_r \). 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

```sql
SELECT * FROM vegetables, supercolumns WHERE vegetables.fied = supercolumns.fied AND vegetables.fid = p_1 OR veg table.fid = p_2
```

The execution time of the above query is insensitive to \( P \), (the number of tuples in \( \text{vegetables} \)), because we place indexes on \( \text{vegetables} \) and \( \text{supercolumns} \). Also notice that for a given value of \( P_r \), \( t_{extract} \) increases with \( P_r \). This is because \( P_r \) is related to the join selectivity of the above query.

Figure 10 shows a plot of \( t_{read} \) versus \( P_r \) for three different values of \( P \). We haven't shown a plot of \( t_{read} \) versus \( P_r \), but we found \( t_{read} \) to be insensitive to \( P \). This is because \( P \) affects \( t_{read} \), in so far as it affects \( P_r \), the number of derived predicates in the Stored D/KB, and as we explained above, \( t_{read} \) is insensitive to \( P \).

Test 3. Study the relative contributions of the different steps in D/KB query compilation time. After making several measurements of the different steps in D/KB query compilation, we found that the principal contributions to \( t_{extract} \) came from the following components:

- \( t_{step} \), the time to set up the query related data structures.
- \( t_{read} \), the time to read the D/KB data dictionary.
- \( t_{reduce} \), the time to generate the evaluation order list, and
- \( t_{compute} \), the time to compute the evaluation order list and link it with the run-time library.

For each query, notice that \( t_{step} \) is insensitive to \( P, P_r \), and \( \text{table} \), because we placed indexes on \( \text{vegetables} \) and \( \text{supercolumns} \).

Also notice that for a given value of \( P \), \( t_{read} \) increases with \( P_r \). This is because \( P_r \) is related to the join selectivity of the above query.

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

Finally, note that the relative contribution of \( t_{reduce} \), the time to compute 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 system. We can make a similar observation about \( t_{compute} \).

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 types of query and data greatly affect the size of the set of relevant facts, \( D \), 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) = parent(X, Y) OR ancestor(Y, Z), ancestor(Z, Y)
```

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 the percentage. Rather, we measured the execution time for naive and semi-nave 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, \( D_{rev}/D_{rel} \), on D/KB query execution time, \( t_e \). We varied \( D_{rev}/D_{rel} \) in two different ways. In the first method, we fixed \( D_{rel} \) and varied \( D_{rev} \) to cover the size of the parent relation fixed and varied \( D_{rev} \) for covering the ancestor query at different subtrees of the parent relation. Thus, each value of \( D_{rel} \) was obtained from a different query, each having a different constant. In the second method, we kept \( D_{rel} \) fixed by fixing the query constant and varied the size of \( D_{rev} \) by making the parent relation progressively larger. Here, the same query was applied to parent relation of different sizes. Semi-nave evaluation was used for LFP computation. Optimization was not used.

Figure 11 shows a plot of \( t_e \) versus \( D_{rev}/D_{rel} \). Notice that when \( D_{rev} \) is fixed, \( t_e \) is insensitive to \( D_{rel} \). 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 \( D_{rel} \) is fixed but \( D_{rev} \) is not, \( t_e \) increases with \( D_{rev} \) (equivalently, \( t_e \) decreases with \( D_{rev}/D_{rel} \)). 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 \( D_{rel} \) fixed and using semi-nave LFP evaluation. We then repeated this procedure for naive LFP evaluation.

Figure 12 shows a plot of \( t_e \) versus \( D_{rev}/D_{rel} \) for both naive and semi-nave evaluation. Notice that for the query and database in this test, naive evaluation is between 2.5 to 3 times faster than semi-nave evaluation. The difference is due to the fact that 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 delay in naive and semi-nave LFP evaluations. There are three main steps when naive and semi-nave 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, whereas the corresponding activities consume 85% of the time in semi-nave evaluation. The activities are not quite the same since semi-nave evaluation computes only the differential of the right hand side of the recursive equations during each iteration.

Test 7 also shows a comparison of the actual time taken to evaluate the right hand side of the recursive equations (or the differential) in semi-nave evaluation and to do the termination check for naive and semi-nave evaluation. Notice that the times for naive evaluation are about 2.5 to 3 times greater than those for semi-nave evaluation. This is the principal reason semi-nave 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 semi-nave, an expensive operation, to be done to evaluate the modified rules and determine the query results against the magic set. In the second part of this test, we executed an ancestor query with very low selectivity (65%), against a parent relation containing over 18,000 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 \( D_{rev}/D_{rel} \) 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 external database is relevant. To understand why, recall that in the magic set strategy, an LFP computation is done first to evaluate the magic rules and then 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. Thus, the results from the first LFP evaluation. When the query selectivity is low, the size of these relationships is small enough so that the 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 these results in a higher overall execution time.

The crossover selectivity where optimization degrades performance for semi-nave evaluation is about 72%, while it is about 85% 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 shows a plot showing the execution times for the two LFP computations as a function of $D/KB$. The rate of decrease 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 $D/KB$, 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 $DKB$, the number of relevant facts, which was varied.

5.3.2 Tests Relating to DKB Updates

This section describes tests to study the effect of $R_k$, $R_w$, and $t_u$ on DKB update times.

Test 8: Study the effect of $R_k$ on the DKB update time. We loaded the Workspace DKB with a single rule and updated the Stored DKB, varying the value of $R_k$ from 9 to 189.

Figure 15 shows a plot of $t_u$ versus $R_k$ 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 $R_k$. The main reason for this is that the time to extract the relevant rules from the Stored DKB is significant with respect to $t_u$ (see next experiment). This insensitivity of $t_u$ to $R_k$ is significant because it means that the DKB 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 $R_k$ and $t_u$. The query update time $t_u$ consists of three major components:

- $t_{u1}$ Time to update the rules relevant to the Workspace DKB from the Stored DKB.
- $t_{u2}$ Time to update the irredundant relations.
- $t_{u3}$ Time to update the irredundant, updates, and irredundant relations.

We updated a Stored DKB with $R_k = 189$ with a Workspace DKB containing 36 rules and measured the values of $t_{u1}$, $t_{u2}$, and $t_{u3}$. The value of $t_{u1}$ was 137, i.e., the transitive closure of the PCG of the Workspace DKB rules after extracting the relevant rules contained 135 edges. We then repeated this procedure for a Workspace DKB containing 1 rule and $t_{u2} = 21$. Table 8 shows the results.

<table>
<thead>
<tr>
<th>$R_k$</th>
<th>$R_w$</th>
<th>$t_{u1}$</th>
<th>$t_{u2}$</th>
<th>$t_{u3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>137</td>
<td>38</td>
<td>189</td>
<td>42%</td>
<td>13%</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>189</td>
<td>81%</td>
<td>6%</td>
</tr>
</tbody>
</table>

Table 8: Breakdown of DKB update time

The main point to note is that $t_{u1}$, the time to extract the relevant rules in a significant component of $t_u$. For small values of $R_k$ and $t_u$, the percentage contribution of $t_{u1}$ decreases, that $t_{u1}$ increases. Also, notice that the time to store the source form of the rules, $t_{u3}$ contributes only a small amount to the overall DKB update time.

6 Conclusions

We have presented the design and implementation of a database knowledge management system architecture and major data structures and algorithms. We also presented and analyzed the results of several experiments carried out to quantitatively measure DKBMS 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.

Rule storage structure design is a major design issue in DKBMS. In our tests, 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 DKBMS 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 DKB. Otherwise, the DKB query execution times will grow with the size of the rule base. The experiment 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 size of the rule 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-space tradeoff. Compiled form rule storage structures like the transitive closure of the PCG use more space but permit faster query computations 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 more than 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 used 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 may take too much time to extract the relevant rules and this can be very time consuming for rules with large PCGs.

As we argued before, from the DKB query computation point of view, a good rule storage structure is one where the relevant rules extraction time is a function of 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 be 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 it. 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, messaging, persistent objects, etc.—with those of DKB query and update processing.

Precompilation of DKB queries can prove to be very useful. Thus, especially true for frequently occurring queries with large $R_k$ values. The price of precompilation is that, for precompiled queries, information about rules and relations 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 $R_k$ values, this price is well worth paying.

Two of the main parameters affecting DKB query execution time are the ratio of relevant facts to total number of facts ($D/KB$). The amount of redundant work done in the LFP evaluation may reduce the amount of redundant work and to restrict the LFP evaluations to the relevant database tuples, the DKBMS 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. As the extra work introduced is negligible and it is possible to tune the DKB query optimizer to adapt the optimization strategy dynamically, switching it on for queries with low selectivity and off for others.

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 whole loop, several table copies are performed. Also, the
5. 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 whole loop execution time.

6. 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, \( r_i = f(r_{i1}, \ldots, r_{in}), \) \( i = 1, \ldots, n \), 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 includes 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 evaluations 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 Shao [16], is one effort along this direction.

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

7. 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 dataflow techniques may be used to evaluate the relational algebra tree corresponding to the right hand side of the equation.

c. Parallel join algorithms may be employed during this evaluation.

8. 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.

We would like to point out that the current version of the testbed is far from a complete D/KBMS. In addition to the implemented components and algorithms (such as the magic sets optimization strategy), there are a large number of 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 Sach, and Allan Anderson.

References

3. 'Database Research at MCC,' Proceedings of the 9th International Conference on Very Large Data Bases, Kyoto, Japan, August 1985.

395