Optimizing the Rule-Data Interface in a KMS

Charles Kellogg Anthony O'Hare Larry Travis

MCC, Austin, Texas U. of Wisconsin, Madison, Wisconsin

Abstract

Work on integrating systems capable of drawing inferences from knowledge bases containing large numbers of logical clauses with relational database systems containing large numbers of facts is described. The aim is to realize the derivational power of symbolic logic while at the same time exploiting the set-processing capabilities and potential parallelism of relational data base systems. We propose that the interface between the deduction and database components involve set-characterizing relational algebra programs (RAPs) and sets of answer values, rather than proceeding sequentially with single answer tuples being requested and returned from the database system one by one. Our design includes a query compiler that translates queries into RAPs, as well as a rule compiler that compiles rules into an efficiently maintainable and incrementally updateable predicate connection graph (PCG), a structure whose use obviates open ended deductive search at query time.

When presented with a query, the system extracts from the PCG a proof schema that represents all possible derivations of the query from the relational database. Structure sharing within this proof schema provides a basis for producing from the schema a significantly optimized RAP. Direct manipulation of the RAP expression enables further optimization, and the optimized program is then evaluated against the database. The result is a set of all possible answers to the query, produced with minimal search of the database. Answers may then be combined with certain intermediate results and proof schema information to generate explanations describing how the answers were derived from the knowledge base.

We describe a prototype implementation of this proposed design and report on preliminary empirical explorations. Some results of the explorations are that, although the number of derivations represented in a proof schema grows log exponentially with respect to deductive complexity (in one example the number approaches three million), RAP size appears to grow only linearly with deductive complexity.

Introduction

Deductive question answering from knowledge bases consisting of large numbers of logical clauses combined with large relational data bases, represents a major requirement for many future computer applications. This paper describes "compilation" techniques for efficiently processing large collections of rules (currently restricted to a pure Horn clause subset of LDL [TZ86]). The compilation consists of transforming the rules into procedural form, as well as constructing a rule/predicate connection graph, that explicitly represents the possible deductive interactions between predicates and rules. Techniques for efficient implementation and maintenance of these structures are described, as are their use for the support of effective set-oriented deductive question answering.

Current logic-based knowledge processing systems typically involve a considerable amount of query-time deductive search. Worse yet, the amount of search required for any particular query is likely to be open ended and unpredictable.

We propose storage representations and processing techniques designed to transfer much of the rule manipulation that constitutes this deductive "search" from query time to compile time, i.e., to the rule entry phase of knowledge processing.

We start from the point of view of deductive question answering based on the predicate calculus and proceed to reduce, where practicable, logical operations to relational-algebra operations. We follow this strategy because we want to maintain the full functionality of logical deduction. In particular, we believe it is essential to preserve the capability inherent in deductive question answering systems to justify or explain answers. Logic-based systems need to present not just conclusions but the justification for those conclusions represented by the structure of inference-rule applications that has led to the conclusions. This justification is one kind of answer "explanation" that is frequently very useful.

An "answer" (other than "yes" or "no") constructed by such question answering systems is an instantiation of the variables in a query where instantiation satisfies a logically valid deduction of the query from the knowledge base (in relational data base theory, such an instantiation is an answer tuple). Our approach is to develop algorithms that produce sets of answer tuples which include members satisfying alternative proofs of a

query. By alternative proofs we mean alternative truth-functional structures (i.e., proof plans) leading to a given conclusion, each of which may itself generate a set of alternative answer tuples. This *proof-schema* or *set-at-a-time*[1] strategy is contrasted with the *tuple-at-a-time* strategies of deductive question answering systems like QA3.5 [GRN69] and logic programming system like Prolog [PRO75]. It also contrasts with the *proof-plan-at-a-time* strategies employed in [KETR76], [REI78], [KUYO82], [JACV84], and [KEL86] which construct data access requests from individual proof plans.

The system combining rule and query compilation that we describe in this paper is implemented currently in fast prototype form. We will call it KCRP (for "Knowledge Compiler Rapid Prototype"). KCRP has been designed to optimize the rule data interface in the following way. For a given query the system generates all possible proof plans and then compiles a single relational algebra program which, when executed, produces the set of all possible answers, as well as information that can be used to explain how those answers were logically produced.

For purposes of attending to first things first and of keeping our prototype manageably simple, we have not directly addressed many issues that would be necessary in a full scale system. For example in real systems with large rule sets and large data bases, complete answer sets could become be so large as to be physically unrealizable (even if they are in theory still finite). User interfaces will have to be developed that enable user control of the relational-algebra program as a generator which will produce on demand desired but limited subsets of the complete answer set. Also our prototype does not yet have the capability for incorporating into its relational-algebra programs the loops that result from recursive rules, although it does include the basis for such an extension. In particular, recursive cycles are explicitly noted and thus easily identified in proof schemas. The KCRP design has placed high priority on algorithmic simplicity and efficiency. For example, a unification-by-exception scheme is employed to minimize calls on a unification procedure. This reflects a major design goal, i.e., to optimize the tradeoff between costly operations such as unification during query processing and complicated schemes for rule storage and maintenance.

Figure 1 illustrates the basic components of the KCRP system. Rules are input to a rule compilation processor which updates the predicate connection graph (PCG). With the representation used for the predicate connection graph, addition and deletion of rules is straightforward, and storage grows linearly with the number of rules. The predicate connection structure directly supports deductive question answering, and simple pattern matching and graph traversal operations replace many of the deductive processing steps required in previous systems. In contrast to other approaches based on Prolog, the system does not force or make use of any ordering of rules or of elements within rules.

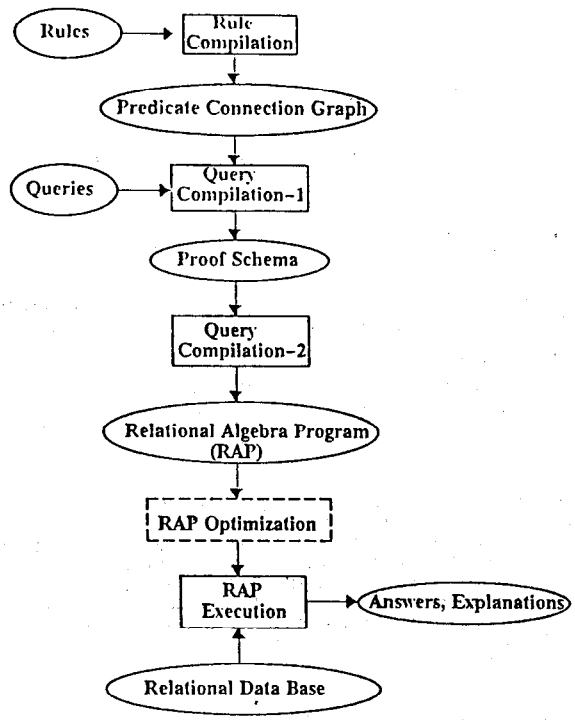

Figure 1. KCRP Components

In processing a specific query, a first-stage query processor extracts from the PCG relevant deductive connections linking the derivable (D) predicates with various rule instances and database (B) predicates whose extensions are contained in the database. The processor recognizes common substructures and produces the set of all possible connections from the D predicate of the query to relevant B predicates. (The prototype works only with queries containing a single predicate but extension to compound queries will not be difficult). In a second stage of query processing, the proof schema[2] is utilized to produce a relational algebra program (RAP) consisting of the standard relational algebra operators, (e.g., union, join, project, and select) and additional procedures to deal with common substructures.

[1]For convenience we often refer to our approach as set-at-a-time rather than the more appropriate but longer phase proof-schema-at-a-time. Clearly our main concern is developing all possible proof plans within the framework of a single proof schema.

[2]A terminological note: We refer to the graph that represents all possible deductive support for a query as its "proof schema". "Proof subschemas" may be extracted from a proof schema, either by pruning away some alternatives under *OR* nodes or by transforming some D-predicate nodes into terminal nodes. A "complete" proof subschema is one all of whose terminal nodes are B-predicate nodes. A distinct "proof plan" is a complete subschema extracted from a schema by pruning away all but one of the alternatives under each OR node of the schema that remains in the subschema. A structure-sharing schema is one where, when there are equivalent substructures, one of them is explicitly represented and all the others are implemented as a "dupe" node that points to that explicit representative. An "essential proof plan" is extracted from a structure-sharing schema and therefore some of its terminal nodes may be dupe nodes.

The RAP produced for a query is highly optimized in terms of minimizing the total number of B predicates and relational algebra operations over these predicates. We do not however currently optimize the RAP execution strategy. This process, shown in dotted lines in Figure 1, requires classical query optimization techniques such as those discussed in [KBZ86]. RAP expressions are evaluated against the relational database, producing the set of all possible answers and a series of "explanation fragments" which may be used to produce as much (or as little) justification of the answers as the user might wish.

Rule Compilation

In this section we briefly review our rule compilation techniques and associated PCG structures designed for efficient representation and maintenance of rules and their deductive associations. We illustrate our ideas with the following rule set for an "invisible college" application:

```
R1: MSIC(S1,S2) <= SU(S1,S3), SU (S2, S3), MT(S3)
R2: MSIC(S1,S2) <= AU(S1,P), AU(SU2,P)
R3: MSIC(S1,S2) <= AU(S1, P1), AU(S2, P2), CI(P1, S2), CI(P2, S1)
R4: MSIC(S1,S2) <= AU(S1, P1), AU(S2, P2), CI(P1,P2), CI(P2, P1)
R5: KNOW(S2, R, T) <= ORIG(S1, R, T), MSIC(S1, S2)
R6: SIF(S1, S2, X, T) <= AT(S1, M, T), AT(S2, M, T), CNF(M, X, T)
R7: KNOWL(L, R, T) <= CRA(S, L, T), KNOW(S, R, T)
R8: KNOW(S2, R, T) <= KNOW(S1, R, T), ABT(R, X),
    SIF(S1, S2, R, T)
```

These rules partially formalize the notion of scientists being members of the same invisible college (clique) in terms of the predicate MSIC. The SIF, KNOWL and KNOW predicates are used to represent knowledge transfer among scientists.[3] These rules provide for deductively augmented access to bibliographic databases consisting of information about scientists, authors, their publications, citation relationships, etc.

Predicate connection graphs have been widely used in mechanized theorem proving where they are dynamically generated during the proof process (see [KOW75], [SIC76], [STI82], [BIB82]). In contrast, we use PCG's as "compiled" structures intended to reduce query-time search. Our PCG's are a knowledge representation (a kind of associative, semantic net) and part of a "permanent" knowledge base (see [KETR81] and [MKSHP81].)

For large rule bases it is crucial to minimize the storage requirements for PCG's. In Figure 2a we show the rule connections for the eight rules presented earlier. Here, one rule node is recursive (R8), i.e., points to itself, as well as being connected to R5 and R6. Substructures, such as that dominated by R5, are shared, as indicated by the links from R7 to R5 and from R8 to R5. Links entering a rule node represent unification paths to the head of the rule, while links leaving a rule node represent deductive interactions between predicates in the body of the rule and the heads of other rules.

Keeping the representation of deductive associations as compact as possible greatly simplifies the process of incremental compilation of the rule base. For example, consider the addition of a new rule

```
R9: KNOW (X,Y,Z) <= KNOW (W,Y,Z),LABA (X,W).
```

Adding this rule to the graph of Figure 2a results in the graph of Figure 2b. Only one additional node has been added but five links have to be inserted to assimilate this one rule.

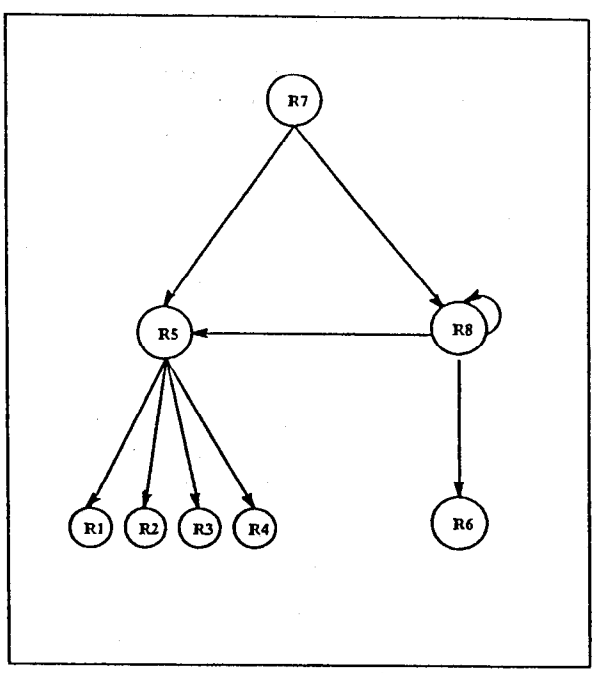

Figure 2a. Connection Graph for eight rules

If we were to represent Figure 2a as an and/or (rule/goal) *tree*, we would require 51 nodes to represent every rule and every goal within a rule as a separate node. Because of the meaning of "tree" , there would be no ability to share common substructures and there would be no ability to represent recursive cycles. If we

[3] Although not important for understanding the abstract structure of the example, the "English" translation of the D predicates is:
MSIC: member same invisible college
SIF : scientific information flow
KNOWL: knowledge by a laboratory of a result
KNOW: knowledge by a scientist of a result
The B predicates have their "English" translation given below:
SU: Studied under
MT: Master Teacher
AU: Author
CI: Cites
ORIG: Originates
AT: Attend
CRA: Conducts research at
ABT: About
CNF: Conference

tried to represent all of the deductive interactions in Figure 2b with such a tree, we would require 125 nodes. If we employed an and/or general graph (i.e., no longer limiting ourselves to use of a tree), recursive cycles and sharing of common substructures would be possible but we would still need 34 nodes to represent each rule and goal explicitly.

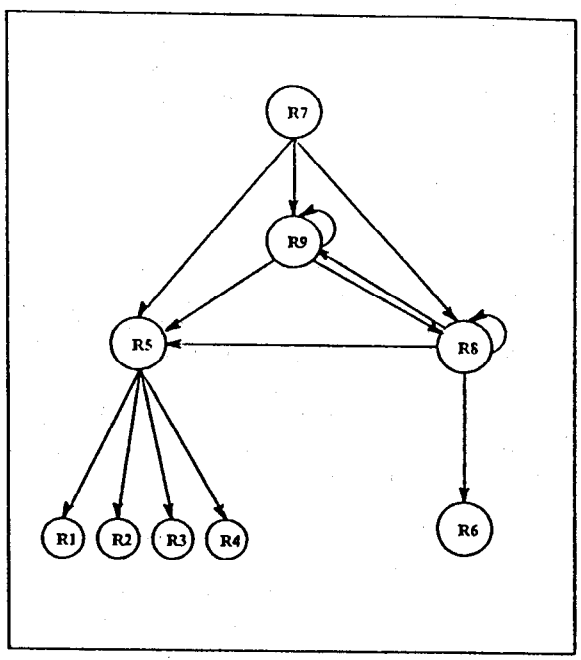

Figure 2b. Connection Graph for nine rules

The PCG structure that we have devised is decomposable into two major parts (in effect, two levels), the rule connection level illustrated in Figure 2a and b, plus a lower level constituted of structures *within* each of the rule nodes. The rule connection level represents deductive associations among rules. The second level represents the compiled procedural form of each individual rule. (The structure used for this internal rule representation is presented below.) This clean separation of inter-rule and intra-rule structure avoids some of the complexity of previous PCG representations where both kinds of information were forced into one composite single-level structure.

Intra-rule deductive associations are represented in a data structure called the predicate rule goal index or, more briefly, rule/goal index. The rule/goal index for the nine rules of our example is shown in Figure 3. A rule/goal index is a tabular array consisting of four or more columns of information. The first column contains predicate pattern identifiers, the second contains predicate patterns where each argument is replaced by "v", "c" or "f" to designate a variable, constant or functor respectively. The third column contains, for each row, a list of identifiers for every rule which has the predicate pattern of that row as its head. For example, in Figure 3, the row for (MSIC vv) indicates this two argument relation occurs as the head of rules R1, R2, R3 and R4. The fourth (the goal) column, contains a similar list of identifiers of rules containing the predicate pattern of that row within their bodies. Again, for (MSIC vv), we can see that it occurs as a goal in the body of R5.

Pred ID	Rule/Goal Index		
	Pred Pattern	Rule	Goal
P1	(AB v v)	-	R8
P2	(AT v v v)	-	R6
P3	(AU v v)	-	R2 R3 R4
P4	(CI v v)	-	R3 R4
P5	(CRA v v v)	-	R7
P6	(CNF v v v)	-	R6
P7	(KNOW v v v)	R5 R8 R9	R7 R8 R9
P8	(KNOWL v v v)	R7	-
P9	(MT v)	-	R1
P10	(MSIC v v)	R1 R2 R3 R4	R5
P11	(ORIG v v v)	-	R5
P12	(SIF v v v v)	R6	R8
P13	(SU v v)	-	R1
P14	(LABA v v)	-	R9

Figure 3. Predicate Rule/Goal Index for Argument/Variable Matching.

Top level goals do not have an entry in the goal column e.g. (KNOWL vvv)), and bottom level predicates (B predicates) do not have entries in the rule column. The number of identifiers in the rule column equals the total number of rules while the number of identifiers in the goal column is equal to the number of distinct occurrences of predicates as goals.

The only work required to add rule R9 to the PCG is to add a new row for the LABA predicate, to insert the R9 identifier into the goal column of the new row, and to insert the identifier into the rule and goal columns of the existing (KNOW vvv) row. Looking at this row we see that there are now three rule identifiers each in the rule and in the goal columns, i.e., 3x3=9 deductive associations between (KNOW vvv) as a rule and (KNOW vvv) as a goal whereas before there were 2x2=4 such deductive associations. The difference of five deductive associations neatly captures the major source of complexity shown in Figure 2b. (If we intersect the head and goal columns for the (KNOW vvv) row we obtain R8 and R9. This indicates that R8 and, R9 are directly and indirectly recursive via (KNOW vvv)). The deletion of a rule is similarly simple and straightforward. We have carefully avoided the use of techniques that require the propagation of effects, such as variable bindings, throughout the PCG during maintenance. One benefit that should be especially useful for maintainers of large knowledge bases is that we can easily compute the precise deductive consequences or *deductive impact* of adding or deleting rules.

The second part of rule compilation consists of compiling each rule into a relational algebra procedure. The procedural form of a rule is represented by a series of joins over predicates in the body of the rule, and a project to obtain the arguments for the rule head.[4] For example, translating rule R8 gives

$$\Pi_{5,3}(\bowtie_{3=5,\ 2=1,\ 1=3} \text{KNOW } (\bowtie_{1=3} \text{ABT SIF}))$$

as the procedural form of the rule.

For a series of Horn clauses where all arguments are variables (such as R1 ... R9) the rule/goal index consists of one row per predicate and one column each for rules and goals. Query constant constraints are pushed down through rules as selects on the rule bodies and unification pattern matching may therefore be avoided.

When constants and/or complex terms occur within rules the rule/goal index is expanded into additional rows and columns as illustrated in Figure 4. Additional predicate patterns for constants (c) and complex terms (i.e., functors "f") are produced when necessary and realized as new rows in the rule/goal index. In addition, rule and goal columns are split into match and unify lists. Rule identifiers in the former are always picked up for a query goal or subgoal that matches the predicate pattern while rule identifiers in the latter may or may not lead to successful unifications. For example if a query goal/subgoal matches predicate pattern P4 (P vc) then rules R1, R3, R4, and R6 will match and rule R5 and/or R7 may unify (for constants unification reduces to a simple, fast equality check). Only in the case of complex terms (e.g., P5, P6) is it necessary to resort to calling a unification algorithm. Term matching thus proceeds very quickly when only constants and variables are involved but term matching is still complete since the full power of the unification procedure (including the "occur" check) is available when needed. The PCG supports not only conventional backward chaining from a query goal toward base predicates, but also forward chaining from a B or D predicate toward its logical consequences, and bidirectional chaining between predicates contained in conditional queries.

R1: P(x,y) ← ...
R2: P(z, f(g(x))) ← ...
R3: P(a, w) ← ...
R4: P(d, y) ← ...
R5: P(y, 125) ← ...
R6: P(w, z) ← ...
R7: P(a, b) ← ...
R8: P(f(x),g(a)) ← ...

Rule/Goal Index			
Pred ID	Pred Pattern	Rule Match	Rule Unify
P1	(P v v)	(R1 . . . R8)	–
P2	(P c c)	(R1, R6)	(R3, R4, R5, R7)
P3	(P c v)	(R1, R2, R5, R6)	(R3, R4, R7)
P4	(P v c)	(R1, R3, R4, R6)	(R5, R7)
P5	(P f f)	(R1, R6)	(R2, R8)
P6	(P v f)	(R1, R3, R4, R6)	(R2, R8)

Figure 4. Row, Column Expansion for Constant and Functional Argument Matching and Unification

[4]There are a number of complications which must be taken care of when we extend our prototype, but they do not cause conceptual difficulties. If several arguments in a predicate expression are represented by the same variable then restrict operators must be employed to force argument positions to be identical. Although this case is handled by the prototype in a locically correct manner, the information is not currently used to constrain database search. Allowance has to be made for variables that occur in the head but not the body of the rule. And relational operator extensions have to be introduced to handle variables that occur in a rule not in a top-level argument place but down within a functor.

Query Compilation-Phase 1

KCRP accepts queries consisting of a single D predicate and its arguments. A proof schema representing all possible proof plans that deductively connect this predicate to B predicates is then constructed. All recursive cycles and common substructures are detected during this process and identified as cycle and dupe nodes respectively.[5] Predicates are not restricted to being exclusively either D or B predicates, i.e., a predicate may have both deductive and database support.

Figures 5a and 5b show the proof schema graphs (for simplicity, base relations are not shown) for minimally and maximally constrained queries involving the KNOW predicate for the sample set of rules. AND nodes are labeled with rule identifiers (e.g., R5, R8, R6) and OR nodes are identified by the predicate identifiers used in the first column of the rule/goal index (e.g., P7.1, and P7.2, label two separate OR structures for the goal KNOW). The query predicate is the root of the proof schema graph, and terminal nodes may be rule identifiers, base relations, boxed nodes, or circled nodes. Under node P7.2 we see a boxed node and a circled node. A boxed node is a pointer to the occurrence of a shared substructure. In this case the substructure headed by rule identifier R5 is shared. Circled nodes represent cycle nodes pointing to recursive substructures. In this case, the recursion returns to the ancestor node R8.1. In this example, there are five essential proof plans.

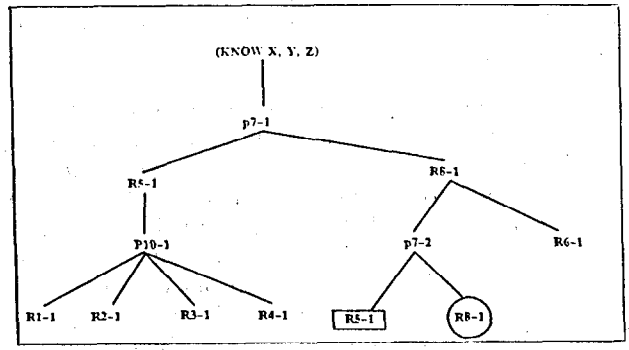

Figure 5a. Proof Schema for "Which scientists know what results when?"

[5] A cycle node in a proof schema is a pointer to an ancestor node whereas a dupe node is a pointer to a non-ancestor, shared substructure.

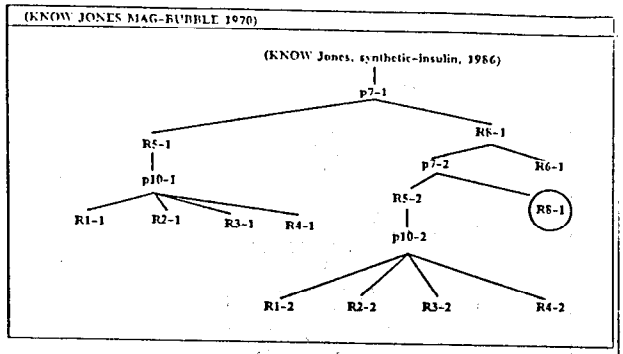

Figure 5b. Proof Schema for "Does Jones know about synthetic insulin results in 1986?"

Figure 5b shows the proof schema graph for the same query except that each of the three arguments is a constant as opposed to a variable. In this case, KCRP generates a considerably different graph. No sharing of substructures is possible because constants are passed down into the graph and substructures that were equivalent for the variable case are no longer equivalent. The result is eight distinct proof plans.

In a proof-plan-at-a-time system, such as KM-1, 50 database literals would be generated for each of the queries in Figure 5. In KCRP for the Figure 5a example, where structure sharing is employed, only eight database literals are generated. For the query in Figure 5b, 12 database literals are generated. Thus we see that even for simple deductive queries involving a small number of rule occurrences there can be a significant savings, when employing a set-at-a-time architecture, in the number of database literals that need to be searched for and retrieved.

Query compilation -Phase 2

This phase of query processing is driven by a proof schema structure. It creates as output a single relational algebra program (RAP) representing all of the extensional database search requirements of the deduction structure developed in the previous compilation phase. The AND and OR nodes in the proof schema are converted into joins and unions in the RAP. The proceduralized rules referenced in the proof schema are nested as required by their deductive interactions and are augmented as necessary with dupe and cycle nodes. A fragment of a KCRP RAP is illustrated below:

```
(MKTEMP 'R8-1 '(KNOWX Y Z)
(PROJECT '(7 1 5)
(JOIN '((2 3)(3 1)(5 4))
(JOIN '((1 2))
(MKTEMP 'DB-6 '(ABT R X)(RETRIEVE 'ABT))
(UNION
(MKTEMP 'R5-2 '(KNOW S R T)(DUPE 'R5-1))
(MKTEMP 'R8-2 '(KNOW S R T)(CYCLE 'R8-1)))))))
```

(Cycle nodes in RAP currently act as no-ops. Recursion is detected, but KCRP does not yet contain the mechanism for translating proof schema cycles into RAP iterators.)

Duplicate nodes, play an important part in avoiding massively redundant data search, since they point to and allow for the reuse of already derived results. One further operation is employed in our RAPs. This is a MKTEMP operation which binds the search results for each rule instance to a temporary relation. These results are used in the later generation of explanations.

The RAP for the query shown in Figure 5a consists of 3 unions, 14 joins, 7 projects, and no selects over eight database literals. The maximally constrained query (5b) employs four unions, 24 joins, 12 projects, and 8 selects (the result of pushing down query constants) over 12 database literals. 528 list cells are required to represent the RAP for the minimally constrained query, and 897 for the maximally constrained one.

RAP Execution, Answer Generation and Explanation

KCRP contains a small relational data management system realized by a series of LISP functions which directly implement select, project, join, and union operations. Additionally, results are reused as required to take advantage of dupe pointers and MKTEMP operations are employed to store intermediate derived relations. In a set-at-a-time system, query optimization can take place at at least two levels. The major form of query optimization employed in KCRP results from proof schema structure sharing. Classical database query optimization techniques are not employed in KCRP currently, but such optimization techniques as pushing selections within joins, and taking joins in the most optimal order (including the reordering of database literals across rules) must be used in a practical system to improve RAP execution.

Even for the sample queries against the sample rule base discussed here quite useful answer justifications may be produced. The justification examples shown below were directly synthesized from the proof schema and the results of evaluating the RAP for the query illustrated in Figure 5a. They have been "Englishized" by hand, to improve the readibility of KCRP Lisp data structures.

Brown, Cook, Davis, Frank, Green, Smith Martin, Patton, Russell, and Jones know about a synthetic insulin result in 1986.

Brown originated the result.

Cook, Green, Smith, Frank and Jones know the result because they are members of Brown's invisible college.

Russell learned about the result from Jones at a conference about the subject of the result during the year of its' origination.

Patton, Green, and Martin learned about the result while co-attending a conference with Smith.

Davis is a member because his scientific publications are involved in a weak citation coupling with Brown's publications.

-
-
-

Results from Experimental Runs with KCRP

Figure 6 presents a series of measures and their values for seven representative queries which have been run on KCRP. The queries designated by columns Q1 and Q2 are the minimally and maximally constrained queries discussed earlier. The remaining five queries are for another application domain involving a larger rule set of 41 rules.

	QUERY						
MEASURE	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Distinct Proof Plans	5	5	306	70,924	54,649	154,744	2,785,392
Essential Proof Plans	5	5	16	628	760	1,053	1,053
Dupe Disjunct	0	0	1	3	3	3	3
Undupe Rules	7	12	13	31	31	31	41
Dupe Rules	1	0	6	31	31	26	24
Recursive Rules	1	1	5	18	18	19	19
Total Rules	9	13	24	80	80	76	85
Dupe DB Literals	10	20	2	11	11	11	14
DB Literals	8	12	5	14	14	14	21
Unions	3	4	7	18	17	16	17
Joins	14	24	4	20	20	20	31
Projects	7	12	8	22	21	22	29
Selects	0	5	0	0	0	0	9
RAP Length	528	897	545	1,855	1,855	1,821	2,300

Figure 6. Statistics for Seven Queries

Figure 7 illustrates a central part of the proof schema for Q7, the most complex deductive query so far processed on KCRP. The longest deductive path (R41 R34 ... R9) consists of 12 rule instances. The pattern of (unioned) rules R12, R16, R18, R20, and R30 occurs five times in this partial proof subschema and the rule pattern (R10, R13, R14, R15, and R17) occurs three times. Both of these *predicate procedure set* patterns are shown enclosed in boxes in Figure 7. Under and overlining are used respectively, to identify dupe and cycle nodes. The number of essential (i.e., with structure sharing) and distinct proof plans is shown to the left of each box (and also for edges leading to other substructures not depicted in Figure 7). The number of proof plans is additive at OR nodes and multiplicative at AND nodes. The multiplicative effect on the number of alternative proof plans can be seen most dramatically for the AND'ed substructures headed by R41, R18, and R13.

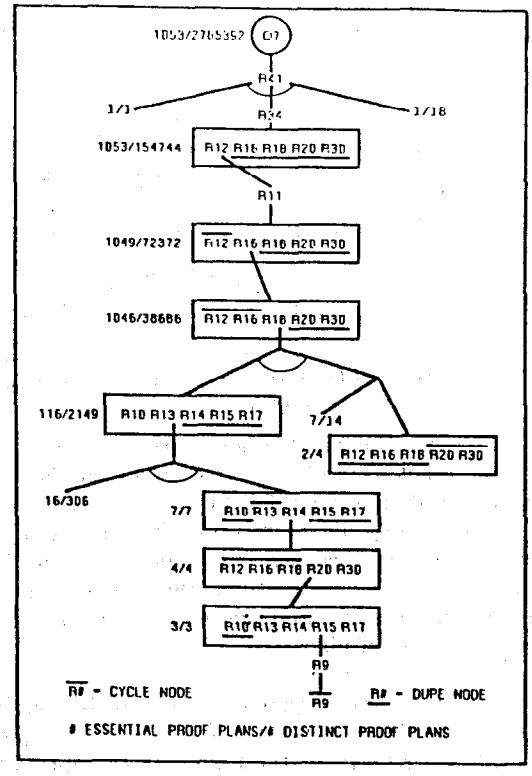

Figure 7. Partial Proof Schema for Q7 Illustrating Recursion and Structure Sharing Patterns

Structure Sharing

KCRP experiments provide dramatic evidence of the importance of structure sharing for a knowledge management system. Information from the first three rows of Figure 6 were used to generate the graph shown in Figure 8a. This graph clearly depicts the log exponential growth of total proof plans vs. the number of original (or "undupe") rule instances employed in a proof schema constructed for a deductive query. We see that the number of distinct proof plans ranges from less than ten for Q1 to almost three million for Q7. For the essential proof plans, made possible by structure sharing, these figures are reduced to five and 1,053 respectively.

By using pointers to common substructures instead of copying or regenerating them we achieve a major reduction not only in the number of rule instances used per

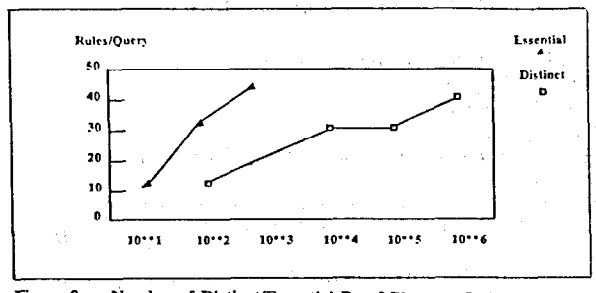

Figure 8a. Number of Distinct/Essential Proof Plans vs. Rules per Query

query but in the number of relational algebra operators used and the number of database literals generated as well. Even where structure sharing does not reduce the number of proof plans, as is the case for the Q2 query, it helps to reduce the number of generated database literals.

The number of database relations per query ranges from five to 21 while the number of database relation duplicates per query ranges from two to 20. Although a RAP containing 21 database relations may seem unusually long, we should note that in this case those 21 relations represent the realization of database search implications for almost three million distinct proof plans. Looked at from the opposite perspective, when a deductive query generates a large number of proof plans as is the case for Q7, it is important to minimize the number of literals that have to be actually evaluated against the database.

So far we have talked about two of the four kinds of structure sharing, duplicate database literals and duplicates of structures headed by rules. We also show the number of duplicates of substructures headed by OR-nodes. This ranges from zero to three in our example. While large numbers of disjunctive duplicates were not found for our set of example queries and rules, where one is found it typically dominates a large substructure and therefore accounts for a significant part of the savings due to structure sharing. The final kind of structure sharing employed in KCRP is subsumption. A given substructure is subsumed by another if the other can be made equivalent to the given one by the application of one or more select operations. Subsumed duplicates for the sample queries range from zero to five per query.

Relational algebra operators

Figure 6 shows the number of union, join, project, and select operations employed in the RAP for each of the seven queries. The most significant result is the combined count of the number of union and join operations required per query. This figure ranges from a low of 11 for Q3 to a high of 48 for Q7. Clearly, for a relational database of any significant size, the processing of these 48 relational operators must be optimized in order to obtain reasonable performance. Query optimization techniques will be an important factor in achieving high performance knowledge management capabilities. When we look at the join and select counts for Q2 and Q7, the two maximally constrained queries, we see the major effects of query instantiation, i.e., pushing query constants thru deductive subgoals into the RAP. Not only are the select counts high, eight and nine, but the join count is also high. The query instantiation process typically results in the generation of more database literals and more unduplicated rule instances than are required when the query does not contain as many constants.

RAP Size

We measure the length of a RAP by the total number of CONS cells, i.e. list nodes, required to represent the RAP in list structure form. Figure 8b shows that the RAP length for the sample queries increases at an approximately linear rate. Clearly, additional experiments are called for with larger rule bases and larger numbers of queries to see if these results hold up. Our initial results suggest that, whereas the number of proof plans tends to increase at a log exponential rate, the increase in complexity of the generated RAP is linear. Note, however, that the deductive complexity, particularly for Q7 which has a total of 85 rule instances in its proof schema, is already fairly deep for deductive question answering purposes, and it is not clear that typical deductive queries for larger rule bases will result in significantly more complex proof schemas. Again, further experimentation is called for .

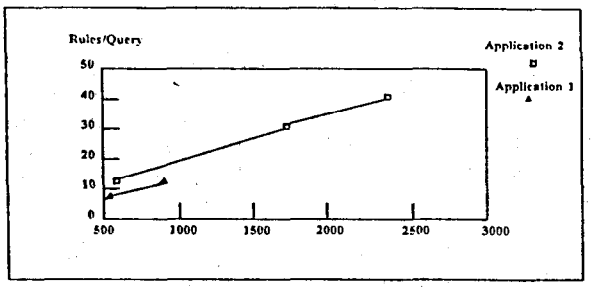

Figure 8b. RAP Length vs. Rules/Query

Extensions and Comparison with PROLOG

We have designed the prototype with hooks for incorporating recursion modules which would replace rule recursion with iteration over data (see [BAN86], [BAN85], [HAN86], [HENA84], [ULL85]). The relational algebra operations we employ must be expanded to include the Extended Relational Algebra operations described in [ZAN85a] for dealing with complex objects and safety tests [ZAN85b] must also be incorporated in order to avoid generating potentially infinite sets.

A comparison of the performance of KCRP and a logic program employing a PROLOG style execution strategy can be made in the following way. For Q7 the logical depth of each of the 2,785,392 distinct proof plans varies between 8 and 12 steps (i.e., logical inferences or unifications). KCRP generates one proof schema and one RAP search request in order to retrieve data that will produce all possible instantiations of those proof plans. A PROLOG depth first backtracking strategy would produce each distinct proof plan a *single plan at a time* and would generate a *separate RAP for each plan.*

Taking PROLOG's backtracking into account we estimate an average of 3.5 unifications per distinct proof plan. Then $2.785 \times 10^6 \times 3.5$ or approximately 9.8×10^6 logical inferences would be required to generate the RAP for

Q7. Since KCRP used 4.6 CPU seconds to process Q7 (on a Xerox 1108) we estimate a PROLOG program would have to run at an effective rate of over two million logical inferences per second (LIPS) to equal KCRP's performance for the query ($9.8x10^6$ divided by 4.6).

Conclusion

Experimentation with the prototype was undertaken to evaluate certain key aspects of the design of a set-at-a-time rule compiling, query compiling architecture. Results have been encouraging and in some cases surprising. Despite considerable experience with logical based KBS's we were surprised to find almost three million distinct nonrecursive proof plans generated for an atomic query. This many proof plans would be prohibitive to deal with either in a tuple-at-a-time or a plan-at-a-time system. Yet in our KCRP we could handle the data base search for the proof schema representing this many plans by retrieving only 21 database literals.

Generation of the PCG rule/goal index has proved to be very fast and the PCG serves as a very compact representation for all possible proof plans using a given set of rules. With the aid of the PCG, proof schemas are quickly generated and Relational Algebra Programs are efficiently derivable from proof schemas.

The set-at-a-time deductive strategy and extensive use of structure sharing provide for almost linear growth in RAP complexity with respect to the number of rule instances used for the proof schema of a query even though proof plans grow at a log exponential rate. It will be important to determine if this continues to be the case for queries involving larger and more complex rule bases.

While RAP length appears reasonable given the above it is still the case that the most complex query produced a RAP containing almost 50 expensive union and join operations. Our experiments indicate that powerful query optimization techniques will be essential for future knowledge base systems.

Our explanation fragment approach provides the basic information necessary to generate useful answer explanations. However, "complete" answer explanations and the compacting and summarizing of explanations will require a substantial amount of additional research.

References

[BAN85] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", MCC, Technical Report, DB-004-85, 1985.

[BAN86] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J., "Magic Sets and Other Strange Ways to Implement Logic Programs", Proc. ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, Cambridge, MA, 1986.

[BIB82] Bibel, W., *Automated Theorem Proving*, Vieweg, Wiesbaden, 1982.

[GMN84] Gallaire, H., Minker, J., Nicholas, J., " Logic and Databases: A Deductive Approach", in *ACM Computing Surveys*, Vol. 16, No. 2, June, 1984, pp. 153-185.

[JACV84] Jarke, M., J. Clifford, Y. Vassiliou, "An Optimizing Prolog Front-end to a Relational Query System", Proc. ACM-SIGMOD Conference, Boston, 1984.

[GRN69] Green, C. C., *Theorem Proving by Resolution as a Basis for Question-Answering Systems*, Machine Intelligence 4, Edinburgh University Press, N.Y. (Meltzer, B. and Michie, D., eds) pp. 183-205, 1969.

[HAN86] Han, J., *Pattern-Based and Knowledge-Directed Query Compilation for Recursive Data Bases.* CSTR #629, University of Wisconsin, 1986.

[HENA84] Henschen, L., and Naqvi, S., "On Compiling Queries in Recursive First Order Databases", in *J. ACM, Vol. 31, No.1, Jan. 1984.*

[KBZ86] Krishnamurthy, R, H. Boral, C. Zaniolo, "Efficient Optimization of Nonrecursive Queries", These proceedings.

[KETR81] Kellogg, C., Travis, L., "Reasoning with Data in a Deductively Augmented Data Base System" in *Advances in Data Base Theory*, Volume 1, Gallaire, H., Minker, J., Nicholas, J., eds., Plenum, 1981, pp. 261-298.

[KETR76] Kellogg, C., Klahr, P., Travis, L., "A Deductive Capability for Data Management", in *Systems for Large Data Bases*, Lockemann, P., and Neuhold, S., eds., North-Holland, Amsterdam, 1976.

[KEL86] Kellogg, C., "From Data Management to Knowledge Management", *Computer*, Vol.19, NBR.1, 1986.

[KOW75] Kowalski, R., "A Proof Procedure Using Connection Graphs", in *JACM, Vol. 22, No. 4, pp. 572-595, 1975.*

[KUYO82] Kunifuji, S., and H. Yokota, "Prolog and Relational Databases for Fifth Generation Computer Systems", in *Logical Base for Data Bases", Toulouse,* December 1982.

[MKSH81] McKay, D., Shapiro, S., "Using Active Connection Graphs For Reasoning with Recursive Rules", *Proceedings IJCAI-81*, Vancouver, 1981.

[REI78] Reiter, R., "Deductive Question Answering on Relational Data Bases", in *Logic and Data Bases*, Gallaire, H., and Minker, J., eds. Plenum, 1978.

[PRO75] Roussel, P., "PROLOG: Manuel de Reference et d'Utilization", *Groupe d'Intelligence Artificielle*, Universite d'Aix-Marseille, Luminy, 1975.

[SIC76] Sickel, S., "A Search Technique for Clause Interconnectivity Graphs", in *IEEE Trans Computers C-25, pp. 828-835, 1976.*

[STI82] Stickel, M., "A Non-Clausal Connection-Graph Resolution Theorem Proving Program", *Proceedings AAAI Conference*, Carnegie-Mellon University, 1982.

[TZ86] Tsur, S., C. Zaniolo, "LDL: A Logic-Based Data-Language", These proceedings.

[ULL85] Ullman, J., "Implementation of Logical Query Languages for Databases" , in *ACM TODS*, Vol. 10, No. 3, Sept. 1985, pp. 289-321.

[ZAN85a] Zaniolo, C., "The Representation and Deductive Retrieval of Complex Objects", *Proc. 11th Int. Conference Very Large Data Bases*, Stockholm, 1985, 1985, pp. 458-459.

[ZAN85b] Zaniolo, C., "Safety and Compilation of Non-Recursive Horn Clauses", *Proc. First International Conference on Expert Database Systems,* Charleston, SC, 1986.