The Interpreted–Compiled Range of AI/DB Systems

Anthony B. O'Hare and Amit P. Sheth
Unisys Corporation

Abstract

A range of approaches to integrating rule-based Artificial Intelligence (AI) systems and Database Management Systems are classified according to the degree of **compilation** *that is performed by the AI system. This* **interpreted–compiled range** *provides a framework for enumerating several relevant aspects of AI/DB systems, for showing how these aspects vary along the dimension, and for comparing and contrasting previous work on AI/DB systems. In particular, this framework focuses on the nature of the interaction between the two systems as well as some of the consequences of the particular approach with respect to the utilization, functionality, and performance of each of the two systems.*

1. Introduction

The integration Artificial Intelligence (AI) and Database Management System (DBMS) technologies promises to play a significant role in shaping the future of computing. As noted in [BROD88], AI/DB integration is crucial not only for next generation computing but also for the continued development of DBMS technology and for the effective application of much of AI technology.

This paper describes various approaches to AI/DB integration in terms of the degree of *compilation* that is performed by the AI component. Note this is but one of many characteristics that one might choose to differentiate the various approaches.

For the purpose of this discussion, it is sufficient to characterize the AI component as a rule-based system that allows a user (or an application) to construct and manipulate a rule base and to pose queries (to the AI component) which entail the firing of *relevant rules* (i.e., those rules that participate in obtaining solutions to the AI query). Further, some of the rules require access to a database that is managed by an external DBMS. The question of where the rule base itself resides will not be considered here.

The term *compilation* in the context of such an AI/DB system refers to the translation of some (possibly all) of the relevant rules into a set of DBMS queries or a program called a *Data Access Program* (DAP), for accessing the database. At one extreme, the *interpreted* approach, the DAP is little more than a simple retrieval operation (i.e., a relational algebra select operation) corresponding to a single database predicate. The results of the DAP are then used by the AI component in further inferencing which may involve the generation of additional DAPs before the processing of the AI query is complete. At the other extreme, the *compiled* approach, the DAP is a complete program for computing all solutions to the AI query.

In this paper, we provide some background for the discussion of the *interpreted–compiled* (I-C) range, present examples for each of the extremes as well as some intermediate approaches, and discuss some of the more significant characteristics of each. A rigorous analysis of the plethora of performance and related issues that are raised here is beyond the scope of this paper. Rather, we are concerned with characterizing and contrasting these approaches in terms of the I-C range.

2. Background

Although logic represents a fundamental approach to the development of rule-based AI systems, there are other approaches such as semantic networks and production systems. However, the vast majority of research into the integration of rule-based inferencing with DBMS systems has taken the logic-based approach. In fact, the notion of the I-C range stems from this research and has not, to our knowledge, been

This work was supported by Defense Advanced Research Projects Agency (DoD) Information Science and Technology Office under contract number N00039-88-C-0100. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

applied to any of the other approaches. The following discussion will be couched in terms of the logic-based approach, i.e., the terminology used here is from the research area known as *logic and databases* (cf. [GALL84]). In particular, we will focus on a general type of AI/DB system referred to as a *deductive database* (DDB).

In general, logic-based systems distinguish between the *intentional database* (IDB), or rule base, which contains general, molecular assertions including integrity constraint rules, and the *extensional database* (EDB), or fact base, which contains only fully instantiated atomic assertions. This does not, however, imply any physical separation of the IDB and EDB. Most implementations of Prolog (cf. [CLOC84]), for example, make no distinction between the IDB and the EDB since both are stored in the same internal database.

Informally, a DDB is an AI/DB system which supports the derivation of new facts from a set of stored facts. Deductive databases that are restricted to first-order clauses are commonly referred to as *first-order databases*. First-order databases are generally restricted to *definite* clauses as opposed to *indefinite* clauses. An *indefinite* clause allows one to express disjunctive conclusions in rules or disjunctive facts. For example, the statement "if John is not at home then John is at work or John is at Jane's house" can be represented directly as an indefinite clause and the statement "John is at home or at work" can be represented directly as a disjunctive fact. A *definite* clause is restricted to a single conclusion (or consequent) as opposed to the disjunction of more than one conclusion that is allowed in the case of indefinite clauses. Hence forth, the discussion of DDBs will be restricted to first-order, definite clause databases.

DDB systems are commonly differentiated on the basis of the query processing strategy that is used. There are two basic strategies or approaches to query processing, *compiled* and *interpreted*. In the compiled approach, access of the EDB is delayed until all the processing required for query evaluation is reduced by the deductive component to a set of EDB accesses (e.g., [FURU77], [CHAN78], [REIT78], [KELL86], [BAKE87], [CHIM87], [OHAR87], [RAMA87], [SCHM87], [SELL88], and [MORR88]).

In the interpreted approach, the deductive component accesses the EDB as it encounters instances of database relations in the process of solving a given AI query (e.g., [MINK78], [CHAK82], [NAIS83], [CHAN84], [LI84], [BOCC86], and [CERI86]).

Although the compiled and interpreted approaches represent the query processing strategies employed by many existing DDB systems, they do not represent all possible strategies. In fact, there are several examples of systems which combine the two strategies (e.g., [KELL78], [KELL81], [BUER85], [IOAN88], and [NUSS88]).

In the following, we view the interpreted and compiled approaches as end-points on a range from little or no compilation to full compilation. Thus, those approaches which mix interpreted and compiled strategies lie somewhere in between the two extremes.

Some of the more salient characteristics of an AI/DB system that are directly influenced by the particular approach that is employed are presented below.

- *DBMS requests*
  The kind of DBMS requests that are generated by the AI component. This may range from a simple retrieval request (e.g., retrieve PARENT where PARENT.#1 = "john") to a complex DAP that can only be expressed in an extended data language (e.g., FAD [BANC87]).
- *Solutions*
  Characterization of the solutions that are produced in response to an AI query and the technique used to produce them. For example, most Prolog implementations are *single-solution* in that they produce one solution to the AI query and generate successive solutions (one at a time) only if requested by the user. In this case, the technique employed involves dynamically expanding the proof-tree for the AI query.
- *DB result granularity*
  The granularity of the result from the DBMS that is actually used by the AI component. Typically this will either be a single tuple of the result relation (i.e., tuple-at-a-time) or the entire result relation (i.e., set-at-a-time).
- *Optimization*
  The kinds of optimizations that are performed or are possible for the given approach. This includes such things as ordering conjuncts (by the system rather than the user) and partial

evaluation. Some optimizations, such as ordering conjuncts, may be performed by the AI component and/or DBMS.

- *Application development*
  There are important implications of each approach on the ability of the system as a whole to support application development. In particular, answer justification and debugging will be discussed.

3. Interpreted Approach

In the extreme case of the interpreted approach, queries to the DBMS are generated whenever a database predicate is encountered in the evaluation of an AI query. For example, consider a Prolog system that is modified so that a database predicate is processed by issuing an equivalent DAP to the DBMS and instantiating any unbound variables with values from the DBMS result (or failing if the result is empty). Given the search strategy of Prolog, this system will make use of only one answer tuple each time the database predicate is encountered and the corresponding DBMS query will be repeated on backtracking (with some variations possible due to different bindings). However, the search strategy also serves to "push selection". That is, constants will be propagated into the database predicates which translate into selection operations within the DBMS.

This system is a single solution one in that it produces one solution to the AI query and attempts to produce another solution only if requested by the user. Such a system may be viewed as building a proof-tree at run-time (i.e., when the AI query is evaluated) containing both IDB and EDB predicates. One proof-tree is constructed for the first solution to the AI query and expanded for each successive solution.

In general, both backtracking and recursion will lead to repeated instances of the same predicates and thus will cause duplicate subtrees of one or more nodes to be added to the proof-tree. Clearly, this redundancy represents an opportunity for improving system performance by reducing or eliminating it. However, the identification of duplicate subtrees is precluded by the fact that the proof-tree is constructed at run-time and no complete history of previous states is retained. Some form of pre-processing of the program (i.e., the AI query and the relevant rules) must be performed in order to support duplicate elimination in general. Alternatively, the problem of redundant DBMS requests could be mitigated to some extent by the use of a buffer or cache within the AI System (e.g., [CERI86]) or between the AI system and the DBMS (e.g., [IOAN88],[SHET89]).

Another consequence of the interpreted approach is that other forms of optimization, aside from reducing redundancy, cannot be performed without substantial modification to the system. For example, the order of conjuncts is determined by the user and can not be re-ordered by the system to improve performance. Even though the DBMS requests consist only of simple retrievals on single relations, the ordering of conjuncts is important since join operations are being performed within the AI component.

The difference between the processing strategies of Prolog and a conventional DBMS gives rise to the well known *impedance mismatch.* That is, the DBMS generates the entire solution set (or result relation) for a DAP while Prolog generates one solution at a time. In general, the DBMS processing strategy can be characterized as *set-at-a-time* using *eager evaluation* since the DBMS operations are applied to entire relations and there is no mechanism for suspending the processing after a single tuple has been produced which satisfies the DBMS query. In contrast, most Prolog implementations can be characterized as *tuple-at-a-time* using *lazy evaluation* since each inference involves a single tuple of a given predicate occurrence and the processing of successive tuples occurs only on backtracking.

A common solution to the impedance mismatch problem is to introduce an interface between the AI and DB components which buffers the results produced by the DB component. For example, [PARK89] describes an interface which uses *stream processing* to buffer the results from the DB component and to pass result tuples, one at a time, as they are requested by the AI component. While such interfaces support a form of lazy evaluation (i.e., a stream will produce a tuple on demand), it is important to note that the entire result relation is produced by the DB component. Thus, the cost of computing the entire result is always incurred by the system even if only a subset of the result is used by the AI component. The use of lazy evaluation in the DB component is discussed in Section 4.

Another approach is to alter the processing strategy of the AI component so that it is set-at-a-time rather than tuple-at-a-time. For example,

[CHAK82] describes a modified Prolog interpreter which processes sets of values instead of individual values. This approach requires deep modification of Prolog's inference mechanism. It also greatly reduces the number of DBMS queries as the full power of relational algebra can be used to increase the bandwidth between the Prolog and the relational DBMS. However, this approach has the drawback of not being able to perform query optimization as in any interpretive approach and, according to [MARQ84], may suffer from an inability to share tables and a limitation on the storage of tables in the implementation.

Application development in the extreme case of the interpreted approach is supported in the same manner as ordinary program development in Prolog. Debugging is supported by via tracing the behavior of the inference engine. While there is no direct support for answer justification, program extension techniques such as those described in [CLAR80] can be employed by the application developer.

In summary, this extreme of the interpreted approach (i.e., accessing an external relational DBMS from a standard Prolog) has the following characteristics.

- *DBMS requests*
  - simple retrieval
  - one user request results in many DBMS requests
- *Solutions*
  - single solution
  - run-time expansion of proof-tree
- *DB result granularity*
  - tuple-at-a-time
  - backtracks for successive tuples
- *Optimization*
  - pushing selection (but ordering of conjuncts is done by the user)
  - lazy evaluation in the DBMS
- *Application development*
  - answer justification via program extension
  - debugging via tracing inference engine behavior

4. Compiled Approach

The compiled approach (e.g., LDL [CHIM87]) attempts to minimize the more complex inferencing operations that will be performed at run-time by "compiling them out", i.e., by performing much of the inferencing at compile-time. If taken to the extreme, the objective is to produce a DAP that need only perform simple matching operations to generate all possible solutions to the AI query. In practice, the DAP can be complex containing operations involving multiway joins and recursion.

Ignoring updates and evaluable predicates, the basic idea underlying the compiled approach is that of constructing a *proof schema* for a given AI query and then compiling out all but the database predicates. A proof schema is an and/or graph where the root is the AI query and the remainer of the graph includes all the rules that are relevant to that query. For example, consider the following set of rules defining the derived relation *grandparent*, where *father* and *mother* are database predicates.

grandparent(X,Y) ← parent(X,Z) & parent(Z,Y)
parent(X,Y) ← father(X,Y)
parent(X,Y) ← mother(X,Y)

Figure 1 shows the proof schema, represented as an and/or graph, for the query *grandparent(a,Y)?* (i.e., find the grandchildren of the individual *a*). Note that all the leaves of the proof schema are database predicates (i.e., *father* and *mother*). In the compiled approach, this graph would be compiled into a DAP that contains only database predicates and join and union operations. Thus, only simple matching is performed at run-time and the more costly unification is eliminated.

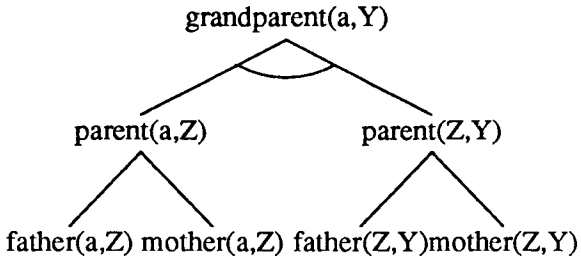

Figure 1

The process of responding to an AI query consists of compiling the query into a DAP and then sending the DAP to the DBMS where all the solutions to the query are computed and finally returned to the user. Since the compilation process can be computationally expensive, the idea of using *query forms* has been used to amortize this cost across multiple queries. In LDL a query form

is a generic query, similar to the notion of mode declarations used in DEC-10 Prolog [WARR77], in which special terms called *deferred constants* are used to denote a term that will correspond to an actual constant in a query. For example, the query form *grandparent($X,Y)*, where *$X* is a deferred constant, indicates that the resulting DAP should be used for any query where the first argument is a constant and the second is a variable. Compound terms containing deferred constants are also allowed, e.g., *p(f(X,$Y),Z)*.

The compiled approach requires that the operations that may appear in a DAP be a superset of those commonly supported by relational DBMSs. For example, in LDL the target language FAD [BANC87] provides operations for iteration (to support recursion); the manipulation of complex and recursive data (to support compound terms); and parameter passing (to support the binding of query constants to deferred constants). [AGRA87] proposes adding an operator called *alpha* to a relational DBMS to support a large class of recursive queries. If a DBMS is used that does not support such operations then an additional layer in the overall AI/DB system is needed which implements them.

Although the compiled approach requires an unconventional DB component, the systems described in the literature adhere to the set-at-a-time execution method used in conventional DBMSs. While this is appropriate for an *all solutions* system, it effectively prevents the system from producing only one solution at a time (as in Prolog). The use of lazy evaluation within the DB component might well provide an efficient technique for supporting such a capability. While this technique has been used in a cache-based AI/DB interface [SHET89], it has yet to be adopted in a fully compiled system. A related technique, which does not rely on lazy evaluation, is *sloppy delta iteration* [SCHM87], which allows the user to force partial solutions to be computed for recursive relations.

A commonly used internal representation for rules is a graph formalism called a *predicate connection graph* (PCG) (e.g., [KOWA75], [SICK76], [KELL78], and [KELL86]). When a query is being compiled, it can be added to the PCG and the resulting graph can be traversed to find all the relevant rules for evaluating a predicate in the query. An optimization that can be done in a query intensive environment is storing the rules in a compiled form as a transitive closure of the PCG (PCG*) [BAKE87]. That is, for each derived predicate *p* all of the predicates reachable from *p* are recorded. This avoids the cost of traversing the PCG for every query to extract the relevant rules.

A related technique referred to as *deep compilation* is described in [SELL88]. In this case, the transitive closure of the PCG for a given predicate is represented as a *Logic Access Path* (LAP). A LAP represents a logical compilation of a given predicate, i.e., it includes an abstract form of a DAP. The LAP's for all of the IDB predicates are integrated into a *Logic Access Path Schema*. One advantage of this approach is that subtrees that are common to different LAPs can be shared. Also, this representation allows the inclusion of results from previous queries to support result-caching. Since the entire rule-base is compiled (at least partially) prior to any query processing, this approach is probably the most extreme of any compiled approach.

The notion of deep compilation raises an issue applicable to the compiled approach in general, i.e., IDB updates. When the IDB is modified by changing a rule, adding a new rule, or deleting a rule, it is necessary to recompile any structure that is affected by the update (e.g., PCG, PCG*, LAP, or DAP). The simplest but most costly solution is to recompile all the structures whether or not they are affected by the IDB update. Use of a selective technique is desirable to minimize the amount of recompilation that will occur when the IDB is updated. For example, [BAKE87] describes an algorithm to compute the transitive closure of the PCG incrementally. That is, whenever the rule base is updated, the transitive closure is performed only on that portion of the rule base that will be affected by the update.

There are a number of other optimizations that can be performed in the compiled approach by virtue of the fact that the "program" (i.e., the AI query and the relevant rules) is processed during a separate phase that is independent of DAP execution. One optimization is a form of partial evaluation which can be done during the construction of the proof schema. In LDL this is referred to as *constant migration* and has the effect of pushing selection but only for non-recursive formulae. The "grandparent" example above shows how the query constant "a" is migrated as far as possible into the relevant rules. Although pushing selection in recursive rules can

not be done at compile-time, there are several recursive query optimizing strategies, e.g., the *Magic Set* and *Counting* methods (cf. [BANC86]), that prescribe methods of re-writing recursive rules so that pushing selection is effectively achieved at run-time. As noted in [BANC86], none of the existing strategies perform well in all cases and some should not be applied in some cases (e.g., the Counting method will not terminate in the presence of cyclic data). Other techniques related to partial evaluation can be applied during the generation of DAP code to improve program efficiency.

The ordering of conjuncts and the selection of a recursive query processing strategy (for a given recursive predicate) represent significant optimizations that can be performed at compile-time. Note that this approach relieves the user from the burden of such optimizations and allows the system to optimize the same set of relevant rules differently for different query forms.

There is also a potential for more global optimizations within the compiled approach. These include *flattening* [KRIS88] across disjunctions, i.e., reformulating a set of disjunctive rules into a set of conjunctions containing only database predicates. For example, consider the following rules where "b1,...,b5" are database predicates.

$p(X,Y) \leftarrow q(X,Z)$ & $r(Z,Y)$
$q(X,Y) \leftarrow b1(X,Y)$
$r(X,Y) \leftarrow b3(X,Z)$ & $b4(Z,Y)$
$r(X,Y) \leftarrow b2(X,Y)$

After flattening, these rules would be as follows.

$p(X,Y) \leftarrow b1(X,Z)$ & $b3(Z,U)$ & $b4(U,Y)$
$p(X,Y) \leftarrow b1(X,Z)$ & $b2(Z,Y)$

There are two distinct advantages of flattening. The first is the reduction of simple deductions (e.g., replacing *q(W,Y)* by its definition) which reduces the amount of computation at run-time. The second, is that it allows the system to order conjuncts across disjunctions. For example, if the optimal odering of the relations "b1, b3, b4" is "b3, b1, b4", this can only be determined via flattening. A significant problem with flattening is the potential "blow-up" in the size of flattened conjunctions.

Since the proof-schema is constructed at compile-time, there is also the potential for identifying duplicate subtrees that can be shared to avoid redundant work [KELL86]. However, it is important to note that such structure sharing restricts the use of *pipelining* [CHIM87] as an execution method. This is due to the fact that pipelining requires that the extension of a predicate can be generated one tuple at a time rather than materialized (as would be the case when the entire relation is created and then saved for later reuse). The relative efficiency or inefficiency of combining pipelined execution methods with the sharing of proof structures is an open question.

Support for answer justification in systems employing the compiled approach has received little attention in the literature. In [KELL86], a rather simple form of support for answer justification is provided by extending the DAP so that a temporary relation is created for the extension of each relevant rule. However, this technique may become unmanageable if the number of rules is very large or the size of the temporary relations is very large. A more selective technique involving the extension of the DAP or of the relevant rules (as mentioned in the interpreted approach) is probably required.

Similarly, debugging has not been directly addressed. However, there are some significant and useful forms of compile-time, program analysis which are relevant to program debugging. For example, *safety analysis* [KRIS88] is used to detect potentially infinite relations. In the case of recursive predicates, the test for safety is based on a sufficient condition and therefor there will be some programs that will fail this test even though they do not, in fact, represent infinite relations.

In summary, the compiled approach has the following characteristics.

- *Solutions*
  - all solutions
  - compile-time generation of proof-schema
- *DBMS requests*
  - complex (e.g., extended relational algebra programs)
  - all relevant DB tuples are accessed and used at once
- *DB result granularity*
  - set-at-a-time
- *Optimization*
  - rule compilation (e.g., PCG* or LAP)
  - incremental rule compilation for updates to rule base
  - partial evaluation (pushing of selection into non-recursive formulae)
  - compile-time ordering of conjuncts

- choice of recursive evaluation strategy
- potential for more global optimizations (e.g., flattening)
- potential for sharing common subtrees
- lazy evaluation in the DBMS when using single solution in the AI system

• *Application development*
- answer justification (via extension of the DAP or of the rules)
- debugging (limited to static program analysis)

5. Variations on a Theme

Somewhere between the interpreted and the compiled approaches are those systems that are capable of compiling more than a single database predicate, but less than an entire proof-schema, into a DAP. Some variations of this approach include limiting compilation to (a) a *proof-plan* (i.e., a subtree of the proof-schema where all but one disjunct in a disjunction have been deleted), (b) conjunctions, and (c) conjunctions composed only of database predicates.

In the idealized version of the first case, a proof-plan is generated at compile-time (as in the compiled approach) which is then compiled into a DAP. Since there are only single member disjunctions in the proof-plan, there is no need to support operations for disjunctions in the DAP language. A variant of this technique is described below.

In general, the proof-plan oriented approach will not yield all solutions since the proof-plan is only a subset of the complete proof-schema. However, it may yield more than a single solution since there may be several solutions obtained from the relevant database predicates. As in the interpreted approach, the decision to search for additional solutions is left to the user.

In the other two cases, compilation is limited to conjunctions of predicates and may be further restricted to conjunctions that contain only database predicates. Whenever an appropriate conjunction is encountered, it is compiled into a DAP which is then sent to the DBMS. The results are then used within the AI component to continue the processing of the proof-schema or proof-plan, as the case may be. Note that the processing is similar in character to the interpreted approach except that this approach attempts to off-load more processing to the DBMS and may make use of set-at-a-time, rather than tuple-at-a-time, operations within the AI component.

In all cases, the execution of the DAP is characterized as set-at-a-time while any processing of the DBMS results may be either set-at-a-time or tuple-at-a-time depending on the implementation.

An approach taken by some (e.g., [CHAK82] and [JARK84]) is to operate a standard Prolog interpreter on a metalanguage program. In this approach, rather than directly evaluating a Prolog predicate by submitting a corresponding DBMS query and receiving the result, the predicate is modified into a form so that its evaluation can be delayed. This process is similar to a "pre-processing" stage in language translation. The modified predicate is then "meta-evaluated". In [JARK84], this involves translating a Prolog clause into a clause in DBCL, an intermediate language, and performing optimizations using relevant integrity constraints. The DBCL is then translated into SQL and submitted to the DBMS. Results from the DBMS are stored into Prolog's internal database where they are processed by Prolog's interpreter. It is assumed that the result from the DBMS will fit in the internal database and that in any Prolog clause, all database predicates are grouped together.

Another variation of the partial compilation approach combines the proof-plan and conjunction compilation techniques (e.g., DADM [KELL78] and the FDE [BUER85]). Such systems employ a technique of delaying or deferring the evaluation of database predicates (also called lazy evaluation) until the entire proof-plan has been processed or some other special condition is encountered (e.g., an evaluable or built-in predicate will force the system to evaluate any previously delayed database predicates). The evaluation of the delayed database predicates is simply the process of compiling them into a DAP, usually as some form of a database join operation, which is then executed on the DBMS. In the event that the evaluation of all the database predicates within the proof-tree can be delayed, the system effectively generates a single DAP for the proof-plan (as in the idealized version described above). Perhaps more typically, several different sets of delayed database predicates will be evaluated before the processing of the proof-plan is completed. Each such set of delayed database predicates may be viewed as a conjunction that is to be compiled for execution on the DBMS.

With the exception of the case where a proof-plan is completely compiled, all of the above may be described as mixing the interpreted and compiled approaches. One consequence of this is that any compilation that is performed must be repeated for similar AI queries and possibly for different proof-plans of the same query. However, one could design a system in which the various DAP fragments could be saved for later reuse.

Another consequence of this mixed approach is related to the question of optimization. Since this approach requires at least enough pre-processing of the program so that conjunctions may be treated as a unit, it is possible to optimize the ordering of the conjuncts in the AI component. However, such optimization will be redundant if the DB component automatically orders joins. In proof-plan oriented systems, other optimizations including partial evaluation and the sharing of common subtrees within the proof-plan are also possible. Further, the use of flattening as an optimization technique is trivial since the proof-plan contains only single-member disjunctions. However, any optimizations can only be applied to a single proof-plan and not to the proof-schema as in the compiled approach. Thus, as with the reuse of previously compiled DAP fragments, redundant optimizations will be performed unless measures are taken to save and recognize previously optimized and compiled portions of proof-trees.

A distinct advantage of the proof-plan oriented technique is the relative ease with which both answer justification and debugging of the application can be supported. Since the proof-plan contains only single-member disjunctions, it is possible to generate DAPs for each conjunction of database predicates so that the information necessary for both answer justification and debugging is available to the AI component. For example, consider the following two rules where all the antecedents are database predicates.

```
msic(X,Y) ← author(X,P1) &
            author(Y,P2) &
            cites(P1,P2)
msic(X,Y) ← studied-under(X,T) &
            studied-under(Y,T) &
            master-teacher(T)
```

There will be at least two distinct proof-plans for the query "msic(X,Y)" since there are two rules defining the "msic" predicate. The DAPs for each proof-plan effectively re-write the rules as follows.

```
msic(X,Y,P1,P2) ← author(X,P1) &
                  author(Y,P2) &
                  cites(P1,P2)
msic(X,Y,T) ← studied-under(X,T) &
              studied-under(Y,T) &
              master-teacher(T)
```

Thus, the AI component can easily determine how a particular result was obtained, e.g., if msic("Jones", "Smith", "Brown") is one of the solutions obtained, then it is clear that the relation msic("Jones","Smith") is true because both "Jones" and "Smith" studied under the same master teacher "Brown". Note that this would not be possible if both rules were compiled in the same DAP since the additional information would be dropped in order to retain the type compatibility of the answer tuples generated by each rule separately.

In summary, the proof-plan oriented, partial compilation approach has the following characteristics. Other approaches involving the mixture of the interpreted and compiled approaches have similar but not identical characteristics.

- *DBMS requests*
  - joins (with select and project operations)
  - all relevant DB tuples are accessed and used at once
- *Solutions*
  - some solutions (more than one is possible but not all)
  - compile-time expansion of single proof-plan
- *DB result granularity*
  - set-at-a-time (within the evaluation of the DAP)
  - tuple-at-a-time or set-at-a-time depending on the AI component
- *Optimization* (limited to a single proof-plan)
  - run-time ordering of conjuncts
  - partial evaluation
  - potential for global optimization
  - potential for sharing common subtrees
  - user directed sloppy delta iteration for recursive queries
- *Application development*
  - answer justification via proof-plan examination
  - debugging via proof-plan examination

6. Conclusion

The I-C range is a scalar dimension along which AI/DB systems vary. In the above, we have enumerated the relevant aspects of these systems, shown how these attributes vary along the

dimension, and placed previous work on AI/DB systems in this framework (Figure 2 illustrates how the various AI/DB integration approaches described above can be ordered along the I-C range).

Previous classifications of the different approaches to AI/DB integration have been based primarily on system architecture (e.g., [GALL83] and [LEUN88]). In [GALL83], four different approaches are identified and distinguished purely on the basis of their constituent components. The taxonomy of deductive database systems presented in [LEUN88] is based on several different categories including *interpreted* and *compiled.* However, as the authors note, there is no clear cut boundary between these categories.

The I-C range provides a simple framework for understanding a variety of possible approaches to the coupling of AI and DB systems. In particular, this framework focuses on the nature of the interaction between the two systems as well as some of the consequences of the particular approach with respect to the utilization, functionality, and areas of potential performance degradation or improvement within each of the two systems.

This I-C range may also be more appropriate for some purposes than other, architecture-based classification schemes. The focus on the interaction between the AI and DB systems should be more useful in the design of alternative system architectures than a framework that dictates the general architectural alternatives.

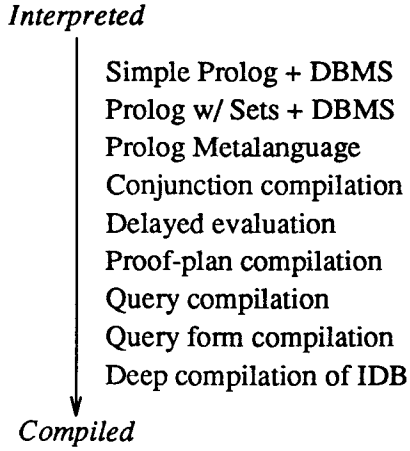

Figure 2

One important question that we have not addressed here is that of the type of AI system that is to be coupled with a DBMS. Specifically, it is not clear that either the interpreted or the compiled approach are appropriate notions in the context of AI systems that are not logic-based. Although an in-depth discussion of this question is beyond the scope of this paper, some general comments can be made. In our discussion of the I-C range we have concentrated on those aspects of AI/DB integration which are to a large extent, independent of the particular knowledge representation scheme employed by the AI component. Most, if not all, of the issues discussed are relevant to the task of AI/DB integration in general. Whether the DBMS requests are derived from Horn clauses, semantic nets, frames, or rules in a production system, they may vary from simple retrievals (e.g., to fill in a slot) to complex DAPs (e.g., to construct instances of some complex object).

7. References

[AGRA87] R. Agrawal, "ALPHA: An Extension of Relational Algebra to Express a Class of Recursive Queries," in *Proceeding of the Third International Conference on Data Engineering*, pp. 580-590, February 1987.

[BAKE87] C. Baker, H. Lu, K. Mikkilineni, R. Ramnarayan, J. Richardson, A. Sheth, and S. Yalamanchili, "Very Large Parallel Data Flow," RADC report F30602-85-C-0125, December 1987.

[BANC86] F. Bancilhon, and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," in *Proc. ACM-SIGMOD 86*, Invited paper, 1986.

[BANC87] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez, "FAD, a Powerful and Simple Database Language," in *Proceedings of the 13th International Conference on Very Large Databases*, 1987.

[BOCC86] J. Bocca, "EDUCE a Marriage of Convenience: Prolog and a Relational DBMS," Third Symposium on Logic Programming, Salt Lake City, Sept. 1986, pp. 36-45.

[BROD88] M. Brodie, "Future Intelligent Information Systems: AI and Database Technologies Working Together" in *Readings in Artificial Intelligence and Databases*, Morgan Kaufman, San Mateo, CA, 1988.

[BUER85] D. Van Buer, D. Kogan, R. Whitney, D. McKay, L. Hirschman, and R. Davis, "FDE: A System for Experiments in Interfaces Between Logic Programming and Database Systems," in *NATO ASI Workshop on Database Machines*, France, 1985.

[CERI86] S. Ceri, G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," in *Proceedings of the 1st International Conference on Expert Database Systems*, Charleston, SC, April 1986, pp. 141-153.

[CHAK82] U. Chakravarthy, J. Minker, and D. Tran, "Interfacing Predicate Logic Languages and Relational Databases," in *Proceedings of the First International Logic Programming Conference*, pp. 91-98, September 1982.

[CHAN78] C. Chang, "DEDUCE 2: Further investigations of deduction in relational databases," in *Logic and Databases*, ed. H. Gallaire, pp. 201-236, New York, 1978.

[CHAN84] C. Chang, and A. Walker, "PROSQL: A Prolog Programming Interface with SQL/DS," Technical Report RJ 4314, IBM Research Laboratory, San Jose, CA, 1984.

[CLAR80] K. Clark and F. McCabe, "PROLOG: A language for implementing expert systems," Technical Report: DOC 80/21, Department of Computing, Imperial College, University of London, November, 1980.

[CLOC84] W. Clocksin and C. Mellish, *Programming with Prolog*, Springer Verlag, N.Y., 1984.

[CHIM87] D. Chimenti, A. O'Hare, R. Krishnamurthy, S. Naqvi, S. Tsur, C. West, and C. Zaniolo, "An Overview of the LDL System," *IEEE Data Engineering*, vol. 10, no. 4, December 1987, pp. 52-62.

[FURU77] K. Furukawa, "A Deductive Question-answering System on Relational Databases," in *Proceedings 5th International Joint Conference on Artificial Intelligence*, pp. 59-66, Cambridge, August, 1977.

[GALL83] H. Gallaire, "Logic databases vs. deductive databases," in *Proceedings of the Logic Programming Workshop*, pp. 608-622, University of Lisboa, Lisbon, Portugal, Albufeira, Portugal, 1983.

[GALL84] H. Gallaire, J. Minker, and J. Nicolas, "Logic and Databases: A Deductive Approach," *ACM Computing Surveys*, vol. 16, no. 2, June 1984.

[IOAN88] Y. Ioannidis, J. Chen, M. Friedman, and M. Tsangaris, "BERMUDA – An Architectural Perspective on Interfacing Prolog to a Database Machine," in *Proceedings of the Second International Conference on Expert Database Systems*, April 1988.

[JARK84] M. Jarke, J. Clifford, and Y. Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System," in *Proc. ACM-SIGMOD 84*, Boston, MA, June 1984.

[KELL78] C. Kellogg, P. Klahr, and L. Travis, "Deductive Planning and Path Finding for Relational Data Bases," in *Logic and Databases*, ed. J. Minker, pp. 179-200, Plenum Press, New York, 1978.

[KELL81] C. Kellogg and L. Travis, "Reasoning with Data in a Deductively Augmented Data Management System," in *Advances in Data Base Theory*, ed. H. Gallaire, J. Minker and J. M. Nicholas, vol. 1, pp. 261-295, Plenum, New York, NY, 1981.

[KELL86] C. Kellogg, A. O'Hare, and L. Travis, "Optimizing the Rule/Data Interface in a Knowledge Management System," in *Proceedings of the 12th International Conference on Very Large Databases*, Kyoto, Japan, 1986.

[KOWA75] R. Kowalski, "A Proof Procedure Using Connection Graphs," in *JACM*, vol. 22, pp. 572-595, 1975.

[KRIS88] R. Krishnamurthy, O. Shmueli, and R. Ramakrishnan, "A Framework for Testing Safety and Effective Computability of Extended Datalog," in *Proceedings SIGMOD 88*, pp. 154-163, 1988.

[LEUN88] Y. Leung and D. Lee, "Logic Approaches for Deductive Databases," IEEE Expert, winter 1988, pp. 64-75.

[LI84] D. Li, *A Prolog Database System*, Research Studies Press, Letchworth, 1984.

[MARQ84] G. Marque-Pacheu, J. Martin-Gallausiaux, and G. Jomier, "Interfacing Prolog and Relational Data Base Management Systems," in *New Applications of Data Bases*, ed. E. Gelenbe, Academic Press, 1984.

[MINK78] J. Minker, "An Experimental Relational Data Base System Based on Logic," in *Logic and Databases*, ed. J. Minker, Plenum Press, New York, 1978.

[MORR88] K. Morris, J. Naughton, Y. Saraiya, J. Ullman, and A. Van Gelder, "YAWN! (Yet Another Window on NAIL!)," *IEEE Data Engineering*, vol. 10, no. 4, December 1987, pp. 28-43.

[NAIS83] L. Naish and J. A. Thom, "The MU-Prolog Deductive Database," Technical Report 83-10, Department of Computer Science, University of Melbourne, Australia, 1983.

[NUSS88] M. Nussbaum, "Combining Top Down and Bottom Up Computation in Knowledge Based System," in *Proceedings of the Second International Conference on Expert Database*, April 1988.

[OHAR87] A. O'Hare, "Towards Declarative Control of Computational Deduction", University of Wisconsin--Madison PhD Thesis, June 1987.

[PARK89] D. S. Parker, "Integrating AI and DBMS through Stream Processing," in *Proceedings of the Fifth International Conference on Data Engineering*, February, 1989.

[RAMA87] K. Ramamohanarao, J. Shepherd, I. Balbin, G. Port, L. Naish, J. Thom, J. Zobel, and P. Dart, "The NU-Prolog Deductive Database System," *IEEE Data Engineering*, vol. 10, no. 4, December 1987, pp. 10-19.

[REIT78] R. Reiter, "Deductive Question-Answering on Relational Data Bases," in *Logic and Databases*, ed. J. Minker, Plenum Press, New York, 1978.

[SCHM87] H. Schmidt, W. Kiessling, U. Guntzer, and R. Bayer, "Compiling Exploratory and Goal-Directed Deduction into Sloppy Delta-Iteration," in *Proceedings of the 1987 Symposium on Logic Programming*, 1987.

[SELL88] T. Sellis and N. Roussopoulos, "Deep Compilation of Large Rule Bases," in *Proceedings of the Second International Conference on Expert Database*, April 1988.

[SHET89] A. Sheth, "Does Loose AI–DBMS Coupling Stand a Chance?," in *Proceedings of the Fifth International Conference on Data Engineering*, February, 1989.

[SICK76] S. Sickel, "A Search Technique for Clause Interconnectivity Graphs," *IEEE Transactions on Computers*, vol. C-25, no. 8, pp. 823-835, August, 1976.

[WARR77] D. Warren, "Implementing Prolog: Compiling Predicate Logic Programs," DAI Research Report No. 40, University of Edinburgh, Edinburgh, May 1977.