4-1991

The Architecture of *BrAID*: A System for Bridging Al/DB Systems

Amit P. Sheth
Wright State University - Main Campus, amit.sheth@wright.edu

Anthony B. O'Hare

The Architecture of *BrAID*: A System for Bridging AI/DB Systems*

Amit P. Sheth
Bellcore
444 Hoes Lane
Piscataway, NJ 08854
amit@ctt.bellcore.com

Anthony B. O'Hare
IBM Corporation
Dept. A42, Bldg. 501
Research Triangle Park, NC 27709
ohare@ralvmm.iinus1.ibm.com

ABSTRACT

We describe the design of *BrAID* (a *Br*idge between *A*rtificial *I*ntelligence and *D*atabase Management Systems), an experimental system for the efficient integration of logic based Artificial Intelligence (AI) and database (DB) technologies. Features provided by *BrAID* include (a) access to conventional DBMSs, (b) support for multiple inferencing strategies, (c) a powerful caching subsystem that manages views and employs subsumption to facilitate the reuse of previously cached data, (d) lazy or eager evaluation of queries submitted by the AI system, and (e) the generation of *advice* by the AI system to aid in cache management and query execution planning. To discuss some of the key aspects of *BrAID* architecture, we focus on the generation of advice by the AI system and its use by the Cache Management System to increase efficiency in accessing remote DBMS through the selective application of such techniques as prefetching, query generalization, result caching, attribute indexing, and lazy evaluation.

1. Introduction

The motivations driving the integration of AI and DB technologies include the need for (a) access to large amounts of shared data for knowledge processing, (b) efficient management of data as well as knowledge, and (c) intelligent processing of data. In addition to these motivations, the design of *BrAID* was also motivated by the desire to preserve the substantial investment in existing databases. To that end, a key design criterion for *BrAID* was to use existing DBMSs as independent system components.

As illustrated in Figure 1 several general approaches have been investigated to support both knowledge processing and data processing. In the following, we briefly discuss this classification to contrast our system with other efforts.

Extending the AI system: In this approach, the AI system is extended with DBMS capabilities to provide efficient access to, and management of, large amounts of stored data. Here the emphasis is on the AI system and only limited DBMS capabilities are added, e.g., [CERI86] implements only the data access layer. Alternatively, a new generation knowledge-based system such as *LDL* [CHIM87] may be developed. Such systems typically provide sophisticated tools and environments for the development of applications requiring intelligence (or, reasoning).

Extending the DBMS system: This approach extends a DBMS to provide knowledge representation and reasoning capabilities, e.g., *POSTGRES* [STON87]. Here we notice the opposite side of the coin from the previous approach. Such systems provide extensive DBMS capabilities, but the knowledge representation and reasoning capabilities are generally quite limited and they lack the sophisticated tools and environments of most AI systems.

Undoubtedly there are applications and environments for which the above approaches are well suited; however, they do not directly support the use of existing DBMSs nor can they directly support existing AI applications (e.g., expert systems) without substantial effort on the part of the application developer.

Loose coupling: The loose coupling approach to AI/DB integration uses a simple interface between the two types of systems to provide the AI system with access to existing databases, e.g., *KEE-Connection* [ABAR86] and *EDUCE* [BOCC86]. The relatively low level of integration results in poor performance and limited use of the DBMS by the AI system. In addition, access to data from the database, as well as the data itself, is poorly integrated into the representational scheme of the AI system. The highly divergent methods of representing data (e.g., relational data models *vs* frames) are generally left to the application developer or knowledge engineer with only minimal support from the AI/DB interface.

Bridging AI and DB systems: The last approach to AI/DB integration represents a substantial enhancement of the loosely coupled approach and provides a more powerful and efficient interface between the two types of systems. As with the previous approach, this method of AI/DB integration allows immediate advantage to be taken of existing DB technologies as well as future advances in them. The problems of performance and under-utilization of the DBMS by the AI system are handled with differing degrees of success first by increasing the functionality of the interface itself, and then if necessary, enhancing either the AI system or the DBMS (usually in this order). However, bridging does ***not necessarily***

*This work was supported by Defense Advanced Research Projects Agency (DoD) Information Science and Technology Office under contract number N00039-88-C-0100 and was performed at Unisys West Coast Research Center and Unisys Paoli Research Center. 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, the U.S. Government or the authors' past or present affiliations.

imply changes to either of the systems. Examples of this type of system are *BERMUDA* [IOAN88] and *IDI* [MCKA90]. Bermuda uses a form of result caching to improve efficiency and performs some simple pre-analysis of the AI application to identify join operations that can be performed by the DBMS rather than the AI system. *IDI* interfaces a hybrid system containing both a frame-based representation and a logic-based reasoning component to a relational database system through a function free Horn clause query language and a simple cahce. Experiments with *BERMUDA* [IOAN90] and *IDI* demonstrate the utility of relatively simple implementation of this approach.

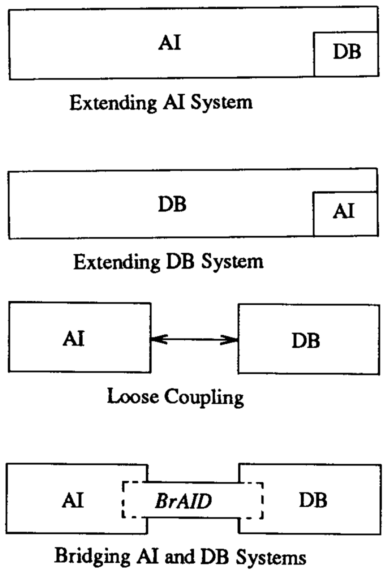

Figure 1: Alternative Approaches to AI/DB Integration

Among the various efforts that might be classified as using either loose coupling or bridging approach to AI/DB integration, most use Prolog (as the AI system) and a relational DBMS (e.g., [MARQ84], [JARK84], [BOCC86], [IOAN88]). Exceptions include the *DADM* [KELL81] and the Flexible Deductive Engine (*FDE*) [VANB85] which consider a number of different inference strategies in addition to that used by most Prolog implementations, and *IDI* (as described in the previous paragraph). While limiting to logic based inference engince, *BrAID* attempts to address issues related to integrating inference engines that vary in the degree of compilation (with Prolog at the interpreted end of the range). Some of the detailed discussions in this paper is however limited to the interpretive inference engine.

BrAID is an effort to further extend the bridging approach and provide higher levels of efficiency and DBMS utilization. Besides significantly enhancing the techniques used in previous efforts such as result caching and performing the pre-analysis of DBMS queries generated by the AI system, *BrAID* employs novel techniques such as *lazy evaluation* and *advice*.

Section 2 discusses the factors contributing to the impedance mismatch between the AI/DB systems and mentions the techniques used in *BrAID* to alleviate the mismatch. Section 3 describes the overall *BrAID* architecture. Sections 4 and 5 discuss the designs of the two *BrAID* subsystems, the inference engine and the cache management system, respectively. This paper gives a general overview of most features of *BrAID* and discusses the generation and use of advice in more detail. Space limitation does not allow discussing all its features as well as implementation of an experimental prototype and some preliminary performance results. These are reported in [SHET89].

2. The AI/DB Impedance Mismatch

The key to efficient AI/DB integration is understanding and alleviating the so called *impedance mismatch* between the two types of systems. This mismatch is the result of differences in (a) the inferencing strategy of the AI system, as embodied in its inference engine, and the data processing strategy of the DBMS, (b) the nature of the queries submitted by an AI system and the functions provided by a DBMS for their efficient processing, and (c) the representation schemes of the two types of systems.

The *interpreted–compiled range* (I–C range) of AI/DB integration [OHAR89b] discusses the first two of these differences in the context of a general taxonomy of AI/DB systems which use a logic-based AI system integrated with a relational DBMS. Since the nature of impedance mismatch varies substantially along the I–C range, it provides a useful framework for our discussion. Specifically, the execution strategy of logic-based systems can be characterized according to the degree of *compilation* that is performed. A fully interpretive system incrementally requests data one tuple-at-a-time (as the need for the tuple arises during inferencing). A fully compiled system compiles that portion of the knowledge base that is relevant to an *AI query* (a request for information given by an application to an AI system), into a single, large DBMS request for a data set which constitutes all solutions to the AI query. Clearly, there is a wide range of possibilities between these two extremes, differing in the scope of data sets requested, in the complexity of query expressions used to specify them, in the frequency of data accesses, and in the optimization performed on the query expressions. An important consideration for designing *BrAID* was to provide efficient integration along several points of this range.

Despite implicit assumptions and explicit claims to the contrary in the literature, it is simply not the case that more fully compiled systems are always preferable. The optimum point on the I–C range will differ with application domains and even from problem to problem within a particular domain. Sometimes results are more useful if provided incrementally. Not all solutions to a problem may be needed or wanted. Important information for optimization of further DBMS requests may not be available at compile time but may become available during inferencing by an interpretive system (e.g., the size of a join extension that turns out to be very much smaller than was estimated prior to actual computation of the extension) -- and thus could be put to good use in reducing the processing cost or response time. Finally, the capabilities of current DBMSs put significant limitations on the feasible degree of query compilation; full query compilation would require substantial extension of their capabilities.

Some of the more salient characteristics of an AI/DB system that are directly influenced by the particular inference strategy include (a) the number and complexity of DBMS requests that are generated by the AI component for a given AI query; (b) how many solutions are produced for a given AI query and the technique used to produce them (e.g., most Prolog implementations are characterized as being *single-solution* in that they produce only one solution at a time while a compiling system such as *LDL*, produces *all solutions*); (c) 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) for an interpretive system and the entire result relation (i.e., set-at-a-time) for a compiling system); and (d) the kinds of optimizations that are performed or are possible for the given approach (it is possible to perform more kinds of optimizations as we go from interpretive to compiling systems).

A primary factor contributing to the impedance mismatch is the AI system's tuple-at-a-time processing and single solution strategy as opposed to a DBMSs set-at-a-time and all solutions strategy. In the Prolog-DBMS integration studies to date, this has been alleviated either by asserting (i.e., transferring the data into Prolog's internal database) result sets produced by the DBMS in response to a query submitted by a Prolog meta-processor (e.g., [CHAN84, JARK84]) or by buffering the result sets (e.g., [CERI86, IOAN88]). Additionally, since both backtracking and recursion lead to repeated instances of the same predicates, a buffer or a cache has been used within either the AI system (e.g., [CERI86]) or between the AI system and the DBMS (e.g., [IOAN88]). However, the use of buffering and caching has been limited to query results (treated as an irreducible unit) and the data is reused only if an exact match of a later query occurs. This limits the extent to which data may be reused, particularly when partial compilation (e.g., conjunction compilation) is allowed in the AI system. By allowing additional processing with the cached data and using a more general subsumption algorithm (see Section 5) than those used previously in AI/DB integration efforts, *BrAID* increases the reusability of cached data. Furthermore, while buffering and caching address the tuple-at-a-time *vs* set-at-a-time mismatch, they do not directly address the single solution *vs* all solutions mismatch. This mismatch is partially solved in *BrAID* through the use of lazy evaluation, i.e., when the query submitted by the AI system can be solved using only cached data then the system produces the result relation incrementally on demand from the AI system. Thus, only those tuples that are required by the AI system will be produced rather than eagerly computing the entire result relation. Clearly, in those cases where the query submitted by the AI system cannot be solved using cached data, then the remote DBMS must be used and lazy evaluation is not possible.

An AI system towards the compiled end of the I–C range uses set-at-a-time processing and typically produces all solutions. Thus there is no real mismatch in terms of the execution strategies used by the two system. However, in this case, the mismatch appears to be due to the differences in the functions required for efficient processing of the usually complex data access programs (DAPs) generated by the AI system and the lack of support of these functions in the DBMS query language and query processing system. Since earlier studies have not tried to interface an AI system towards the compiled end of the range with an external DBMS of the types available commercially, this mismatch has been largely ignored. While queries generated by an interpretive AI system are simple (albeit more frequent), the DAPs generated by a compiling AI system are quite complex, often involving union, recursion and/or iteration, and complex data structures. In *BrAID*, we propose to use second-order templates along with specialized operators (e.g., a fixed point operator) to alleviate much of this mismatch.

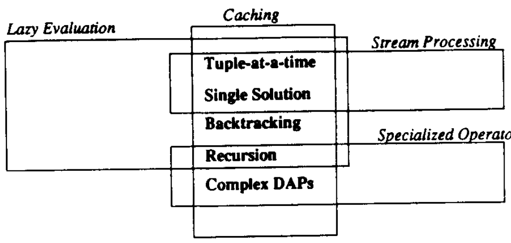

Figure 2: Alleviating the Impedance Mismatch

Figure 2 summarizes some of the more significant techniques utilized in *BrAID* to alleviate specific aspects of the impedance mismatch. Each rectangle in the figure represents one of the techniques (shown in *italics*) and encloses those aspects of the impedance mismatch (shown in **bold** face) that are alleviated by the given technique (e.g., caching, as used in *BrAID* is relevant to all of the aspects listed while stream processing is most relevant to only the first two aspects).

3. Architecture of *BrAID*

The additional capabilities needed to alleviate the AI/DB impedance mismatch can be either provided by extending a DBMS or in the interface between the two systems. While we expect that next generation DBMS will provide some of the capabilities that will alleviate this mismatch, no one has proposed a DBMS that would have all the capabilities needed for alleviating all of the aspects of the mismatch that appear when integrating a range of AI systems with a DBMS. Also, including some of the additional capabilities in a DBMS may be useless for other non-expert system based applications that share the database and may introduce significant system complexity and inefficiency. And, in our case, the use of a conventional relational DBMS was a requirement. Hence the additional capabilities required to alleviate the mismatch are provided by an interface subsystem called the Cache Management System. This resulted in the general architecture for *BrAID* shown in Figure 3.

BrAID consists of three major components, an inference engine (IE), a Cache Management System (CMS), and a remote DBMS. The first two are realized on a workstation and the third is realized on a separate system (database server). The role of the CMS is to facilitate interaction between the IE and the remote DBMS, enabling full and effective use of the DBMS at low cost -- where cost is measured in terms of volume of communication between the workstation and the remote system, computational demands made on the database server, and computation that needs to be done by the workstation (which may be measured as the response time).

The user or application submits an AI query, which is an atomic formula in first order logic, to the IE. The IE then attempts to provide solutions to the AI query by drawing deductive inferences from the rules in its knowledge base combined with data from the DBMS. The IE interfaces with the CMS using a well defined interface consisting of the *C*ache *Q*uery *L*anguage (CAQL) (see Section 5) and the advice language (see Section 4). CAQL is more general than SQL. Database access by the IE is represented in terms of CAQL queries (i.e., the DAP consists of one or more CAQL queries). We recognize that the IE will have direct access to the problem specific information highly relevant for efficient management of the cached data and efficient processing of future CAQL queries. This information is expressed as an expression in the advice language.

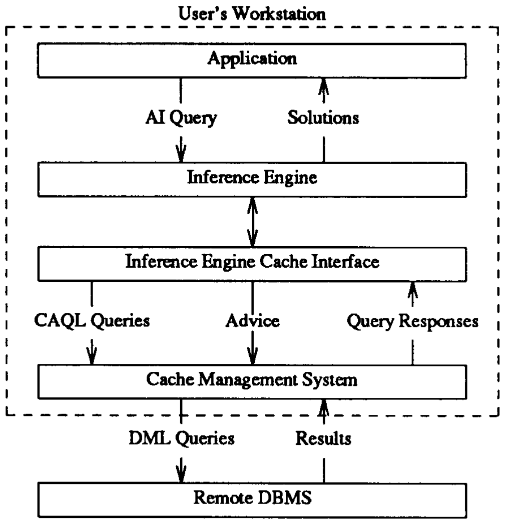

Figure 3: *BrAID* Conceptual Architecture

Functionally, the CMS is a main memory relational database management system where the database, referred to as the cache. The cache consists of relations which are typically views over the remote database as defined by CAQL queries. This contrasts with buffering of only single relation extensions (e.g., [CERI86]) or views defined by joins [IOAN88]. The CMS accepts CAQL queries and advice from the IE and executes CAQL queries by accessing data from the cache and/or the remote DBMS. The CMS is functionally more powerful than a traditional DBMS. It employs a subsumption algorithm to find all relevant data in the cache for a given CAQL query. To retrieve data from the remote database, it performs query translation to data manipulation language (DML) of the remote DBMS.

The typical mode of IE – CMS interaction consists of a set of sessions. At the beginning of each session, the IE submits a set of advice. This is followed by a sequence of CAQL queries. The CMS returns the result for the query using a stream.

The division of labor among the IE, the CMS, and the remote DBMS can further be understood by noting the primary information for which each is responsible: The IE controls the knowledge base, the CMS controls the cache and the cache model (i.e., meta-information about the cache), and the remote DBMS controls the database and the database schema.

In *BrAID*, a component may query other components through a well defined interface. This ability to query is primarily from top to bottom (with respect to Figure 3). Thus the IE can access cache model information from the CMS, and it can access the schema information from the DBMS (via the CMS), and the Cache Manager can access any information that the DBMS provides. However, since the DBMS is treated as an independent system component, it does not access any information from any other *BrAID* component. Also the information that is controlled by the IE is available to the CMS only via the advice provided by the IE, i.e., the CMS only receives advice and does not actively request it, *nor is advice necessary* for the CMS to function. Thus the CMS may be used by systems other than the one described here.

4. Inference Engine

In general, work aimed at enabling an AI system to make use of a DBMS has been after the fact. That is, the AI system has been designed to work in isolation, e.g., on a Lisp machine, and the DBMS schema has been designed without any special consideration given to demands that will be put on the DBMS by the AI system. Then an attempt has been made to patch the two together. Consider, for example, the many attempts that have been made to put Prolog programs in front of DBMSs. While a powerful interface (such as the CMS) can provide higher efficiency than have been shown in previous efforts, we believe that the integration can be made substantially more efficient by designing the AI system with efficient DBMS utilization in mind. The algorithms that the AI system realizes and the data structures it uses need to be specially tailored. One of our primary goals has been to discover useful forms for such special tailoring. The following are some of the more important aspects of our approach for the IE design.

Realizing various inferencing strategies: *BrAID*'s IE does not use a built-in inferencing strategy (also called deductive search strategies). Rather, it makes available a set of component functions that can be combined into various tailored "function suites" as done in the *FDE* [VANB85] to effect several different strategies along the I–C range. This is for two reasons: it enables us to *gain experience with DBMS access problems as they vary over inferencing strategies*, and it is a step toward the long run goal of realizing an inference system capable of adapting its choice of inference search strategy to the problem at hand.

Advice Generation: Advice is a specification of problem specific information useful in optimizing remote database access. Essentially, advice from the IE provides the CMS with predictions about how the IE will be accessing database relations during the course of solving a given AI query. Such predictions range from the very precise *view specifications* to the more abstract *path expressions* described in Section 4.2.

Eager constraining of the problem graph: Inferencing involves finding patterns in a problem graph, a graph representing that part of the AI system's knowledge base relevant to a particular AI query. A general, unimpeachable AI heuristic is to use all available knowledge to constrain the search space. This takes the form of constraining the problem graph as early as possible. As discussed in the next subsection, *BrAID*'s IE will construct the graph for the problem, and prior to undertaking systematic traversal of the graph and systematic querying of the DBMS, it will do everything possible to cull and constrain the graph.

Use of second-order properties: In addition to the first-order expressions typically contained in a logic-based knowledge base, we include in our knowledge base limited kinds of second-order assertions (SOA's), in particular, mutual exclusion and functional dependency SOA's useful for problem graph culling and constraint, and SOA's that define certain relations as recursive structures of other relations (cf. [OHAR87]).

Since a detailed treatment of all of the above aspects is beyond the scope of this paper, following a brief discussion on the IE organization, we will focus on the generation and use of advice. Similarly, the discussion of the IE will be restricted to a single inference strategy, i.e., the well-known depth-first with chronological backtracking strategy of Prolog.

4.1. Inference Engine Organization

The activity of the IE can be broken down into six subtasks as illustrated in Figure 4. Each of these subtasks is delegated to and performed by a module designed specifically for the subtask. We now briefly discuss the responsibilities of each module except the *query translator* which simply accepts the AI query and translates it into the internal form.

Problem graph extractor: The problem graph extractor extracts from the predicate connection graph that subgraph based on rules and second-order knowledge relevant to the AI query [KELL86]. A problem graph is an and/or graph consisting of alternating levels of AND nodes and OR nodes. An AND node represents a rule, i.e., the AND node represents the head of the rule and its successors (which are anded together) represent the antecedents in the body of the rule. Each antecedent is represented by an OR node. An OR node contains a single relation occurrence (or subgoal) and its successors form a subgraph that represents the different clauses (rules) that define that relation.

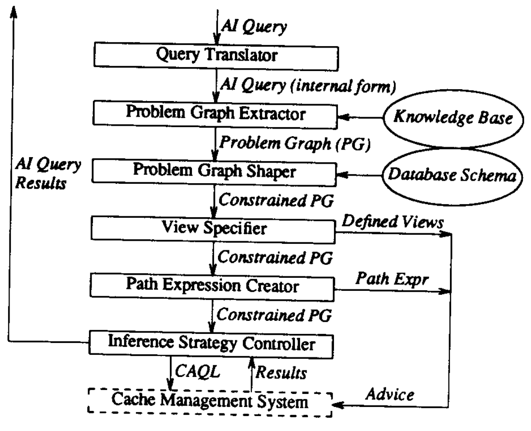

Figure 4: Inference Engine Organization

Problem graphs are constructed by performing partial evaluation of an AI query. This is similar to the process of interpreting a logic program. The main difference is that, in general, the evaluation procedure is applied only to relations that are user-defined and not to database relations or to built-in relations (e.g., arithmetic or numeric comparison relations). Thus, the problem graph is a partial proof-tree for the query where the leaves of the graph are either database relations or built-in relations. Note that recursively defined relations represent a special case. Although they are user-defined, only a single instance of the recursive definition will appear in the subgraph for each recursive relation occurrence.

Problem graph shaper: The problem graph shaper eagerly constrains the problem graph using constant propagation techniques. For example, constants from the AI query and from the parts of the knowledge base represented in the problem graph are pushed along variable sharing and unification arcs as far as possible. Such constants may also be produced by evaluating predicates all of whose arguments are bound (or known) at this stage of processing. In addition, cardinality and selectivity information from the DBMS schema and from functional dependency SOA's in the knowledge base is used to determine producer–consumer relationships (which gets translated into conjunct orderings -- if the IE is free to re-order -- and into binding patterns). Finally, parts of the problem graph under OR nodes are culled away to the extent that this is logically valid given its constant pushing and mutual exclusion SOAs in the knowledge base.

View specifier: The view specifier flattens a problem graph (i.e., performs a constrained process whereby parts of the problem graph are converted to disjunctive normal form) and produces a set of *view specifications* (see Section 4.2.1). Parameters control the extent to which flattening is applied. Sequences of base and evaluable predicates under an AND node constitute a candidate for a view specification. As with flattening, a parameter controls the maximum size of the conjunctions that can be transformed into view specifications (with 1 being the smallest possible value).

Path expression creator and inference strategy controller: The path expression creator constructs a *path expression* (see Section 4.2.2) by traversing the problem graph. All alternatives under decision points must be traversed because the path expression creator will not have available the DBMS contents on which the decision will be based when actual inferencing is being done. Once the path expression has been transmitted to the CMS, the inference strategy controller systematically walks the problem graph and sends CAQL queries in order to solve the problem posed by the original AI query.

Next we discuss the advice language and the types of information advice could provide to the cache management system and breifly discuss the generation of advice (more complete details can be found in [OHAR89a]).

4.2. Advice Language and Advice Generation

There are two primary types of advice that are generated by the IE: *view specifications* (or relation definitions) and *path expressions*. Each of these advice types can range from simple to complex by omitting or adding various features.

The simplest kind of advice that the IE can provide to the CMS will be an unordered list b_1, b_2, b_3, ... , of all the base relations referenced in the problem graph, i.e., a list of all the base relations relevant to the IE's current problem according to the rules contained in the knowledge base. While the kinds of advice presented below are considerably more robust and richer in terms of the relevant information that can be conveyed to the CMS, it is important to note that even this simplest form of advice will provide the CMS with significant knowledge about an AI query.

Advice, particularly, path expressions, can be used by the CMS in making the following critical decisions.

Prefetching: which data accesses (if any) should be made with their results stored in the cache prior to receiving a CAQL request involving them. Also, *when* such prefetching should be performed.

Result caching: should the results of a CAQL query be cached or not.

Replacement: which contents of the cache are the best candidates for replacement (in the event that the cache becomes full and some previously cached relations must be discarded).

Attribute indexing: which (if any) of the attributes of a cached relation should be indexed to increase the performance of random accesses.

Cache *vs* DBMS execution: which parts of a CAQL query should be executed locally by the CMS and which parts of a query should be executed on the remote DBMS.

Lazy *vs* eager evaluation: which parts of a CAQL query should be evaluated lazily and which parts should be evaluated eagerly.

Generalization: should a CAQL query be generalized (i.e., should constants in the query be replaced with a more general form such as variables or ranges of values).

4.2.1. View specifications

The simplest kind of view specification will be unmodified base relations, base relations constrained with binding patterns, and selections and projections on base relations, as these are easily determinable from the argument strings (after constant pushing) of relation occurrences in the base relation fringe of the problem graph. Of interest for present purposes is the determination of what joins should be used as the basis of view specifications. Selection and projection operations will of course be combined with join operations in the same definition, but in the following we focus on joins *per se*. This kind of advice will thus consist of view definitions similar to those used in traditional relational database systems.

There is a direct mapping between view specifications and CAQL queries produced by the IE. That is, any given CAQL query will necessarily be a single view specification with zero or more query constants and/or variables.

The general form of a view specification is:

$$d_i(\ldots) =_{def} c_j(\ldots) \mathbin{\&} \ldots \mathbin{\&} c_n(\ldots) \quad (Rj,\ldots,Rk)$$

where $d_i(\ldots)$ is the defined relation, $c_j(\ldots)$, ..., $c_n(\ldots)$ are cache elements that can be processed by the CMS, and *(Rj,...,Rk)* is a list of rule identifiers indicating the source of the relation occurrences used in the view definition. A cache element is defined by a CAQL expression and may be a base relation, a view, or an evaluable function. The rule identifiers are added here for human consumption rather than for use by the CMS. However, such information will be of use within the system when the problems of debugging and answer justification are addressed.

The argument set for a d_i is the minimum set of variables required for the deduction in which it appears. In general, simply using the set of all variables appearing in the expressions of the cache elements $c_j(\ldots)$ through $c_{\backslash f2n}(\ldots)$ will introduce extraneous variables. The minimum argument set, **A**, of a d_i is given by the formula- $\mathbf{A} = [\, \mathbf{H} \cup \mathbf{B} \,] \cap \mathbf{D}$
where **H** is the set of variables appearing in the head of the current rule (from which the d_i definition is derived); **D** is the set of variables appearing in $c_j(\ldots)$ through $c_n(\ldots)$; and **B** is the set of variables appearing in the body of the current rule (after the relations forming the $c_j(\ldots)$ through $c_n(\ldots)$ have been deleted). For example, given the following rule where b_i reperesents a base relation and k_i represents a releation defined by some rule(s)
$k_9(X,Y) \leftarrow k_2(X,Z) \mathbin{\&} b_1(Z,W) \mathbin{\&} b_2(W,U) \mathbin{\&} b_3(U,V) \mathbin{\&} k_3(V,Y)$
the view definition would be
$d_i(Z,V) =_{def} b_1(Z,W) \mathbin{\&} b_2(W,U) \mathbin{\&} b_3(U,V)$

Since every occurrence of a d_i is unique, it is possible to augment the relation definitions with *consumer* and *producer* annotations, i.e., indicators of which arguments of a corresponding CAQL query will be constants and which will be variables. The following are some examples of binding annotations on relation definitions.
$d_1(X\hat{},Y?) =_{def} b_2(X\hat{},Z) \mathbin{\&} b_3(Z,Y?)$
$d_2(X?,Y?) =_{def} b_1(X?,c_1) \mathbin{\&} b_2(X?,Z) \mathbin{\&} b_3(Z,Y?)$
Here, $X\hat{}$ indicates a free variable (i.e., executing the corresponding CAQL query will produce a set of bindings for it) and $Y?$ indicates a bound variable (i.e., the corresponding CAQL query will have a constant in place of $Y?$). Note that variables which appear only in an antecedent are not annotated since the CMS will be responsible for ordering the antecedent and any annotation would imply a specific ordering (in order to preserve the implied produce/consumer relationships).

A significant use of such annotations (by the CMS) is for indexing decisions. That is, the consumer annotation ("*?*") constitutes advice to the CMS that the given attribute in the given relation occurrence is a prime candidate for indexing while a producer annotation ("^") constitutes advice against indexing. If a given relation is strictly a producer relation (i.e., none of its attributes have consumer annotations) then the CMS will be well advised to produce the relation lazily and without any indexing. It may also choose not to cache the relation if there are no other predicted requests for it.

4.2.2. Path Expressions

A *path expression* represents a prediction of relation accessing order, repetition, and binding patterns. In the ideal case where the IE could possess perfect self knowledge, it could generate a path expression that would represent the ordered set of all CAQL queries that it will emit during a session (i.e., the processing of a single AI query). This would allow the CMS to manage the cache perfectly for a whole session since it could predict precisely what relations would be needed at any given time. Although this is hardly realizable, it provides us with an idea of the goal of this type of advice. Path expressions are an *abstraction* of the CAQL query sequence that will be emitted by the IE during a session. We use the term *query pattern* to denote an abstraction of an individual query. The closer that abstraction is to the actual output of the IE, the better the CMS will be able to plan query executions and manage the cache. Path expressions include abstractions of CAQL queries, constants, and both sequences and alternative sets of CAQL queries.

The primary component of a path expression is the *path expression element* which may be either a single *query pattern* or a grouping of one or more path expressions which are sub-expressions of the enclosing path expression. A query pattern has the general form $d_i(T_1,...,T_n)$ where d_i is the identifier of a view specification, T_j is a variable or a constant, and $n \geq 0$.

A grouping may be either a *sequence* (denoted by the grouping operators "( )") or an *alternation* (denoted by the grouping operators "[ ]") of path expressions. A sequence represents a precise ordering of its member path expressions with respect to the CAQL queries that will be emitted by the IE during the processing of the current AI query. Associated with each sequence is a *repetition count* which provides a lower and upper bound on the number of times the sequence will be repeated. An alternation represents an unordered set of path expressions that will be emitted by the IE during the processing of the current AI query. Of the members of the alternation, one or more may be emitted by the IE but the order is unknown and some members may never appear at all. Associate with each alternation is an optional *selection term* which is a positive integer indicating the maximum number of elements that may be selected during any occurrence of alternation.

Example 1: The following example is based on the following rules and the AI query "$k_1(X,Y)?$ ".
R_1: $k_1(X,Y) \leftarrow b_1(c_1,Y) \,\&\, k_2(X,Y)$
R_2: $k_2(X,Y) \leftarrow b_2(X,Z) \,\&\, b_3(Z,c_2,Y)$
R_3: $k_2(X,Y) \leftarrow b_3(X,c_3,Z) \,\&\, b_1(Z,Y)$
The view specifications for the AI-query "$k_1(X,Y)?$" would be:

$$d_1(Y\hat{}) =_{def} b_1(c_1,Y\hat{}) \qquad (R_1)$$
$$d_2(X\hat{},Y?) =_{def} b_2(X\hat{},Z) \,\&\, b_3(Z,c_2,Y?) \qquad (R_2)$$
$$d_3(X\hat{},Y?) =_{def} b_3(X\hat{},c_3,Z) \,\&\, b_1(Z,Y?) \qquad (R_3)$$

and the path expression would be:
$$(d_1(Y\hat{}),\, (\, d_2(X\hat{},Y?),\, d_3(X\hat{},Y?)\,)^{<0,|Y|>}\,)^{<1,1>}$$

The path expression in example 1 indicates that CAQL queries generated by the IE in solving the AI query "$k_1(X,Y)?$" will be "$d_1(Y)$" possibly followed by "$d_2(X,c)$" possibly followed by "$d_3(X,c)$" where c will be some constant value from the set of bindings for Y in the first query. There will be at most $|Y|-1$ recurrences of $d_2(X,c)$ possibly followed by $d_3(X,c)$ where $|Y|$ is the number of unique bindings for Y. No additional "$d_1(Y)$" queries will occur since the repetition term is $<1,1>$.

In order to understand why there will be only a single $d_1(Y)$ query emitted by the IE, it is necessary to understand the nature of the interface between the IE and the CMS (see Section 5). The result of the query $d_1(Y)$ will be a stream of zero or more tuples which are produced by the IE one at a time. Thus, it may be thought of as repeating the query $d_1(Y)$ for each tuple in the stream. Such repetitions are performed *internally* by the IE and are not emitted as explicit CAQL queries to the CMS.

The following example is similar to the previous one except that alternation is required to describe the modified definition of the k_2 relation. The AI query and first rule are identical to those above, only the second and third rule are changed, i.e.,
$R_{2'}$: $k_2(X,Y) \leftarrow k_3(X) \,\&\, b_2(X,Z) \,\&\, b_3(Z,c_2,Y)$
$R_{3'}$: $k_2(X,Y) \leftarrow k_4(X) \,\&\, b_3(X,c_3,Z) \,\&\, b_1(Z,Y)$
Occurrences of $k_3(X)$ and $k_4(X)$ are to be processed entirely by the IE. If $k_3(X)$ in rule $R_{2'}$ fails then no CAQL query involving the database relations b_2 and b_3 will be emitted by the IE. Similarly, if $k_4(X)$ in rule $R_{3'}$ fails then no CAQL query involving the database relations b_3 and b_1 will be emitted.

The view specifications for this example would be the identical to those of the previous example and the path expression would be:
$$(d_1(Y\hat{}),\, (\,[d_2(X\hat{},Y?),\, d_3(X\hat{},Y?)]\,)^{<0,|Y|>}\,)^{<1,1>}$$
The difference between this and the previous path expression is that the query $d_1(Y)$ may be followed by either $d_2(X,c)$ or $d_3(X,c)$ instead of only $d_2(X,c)$.

Path expression tracking deals with the problem of establishing an association between a given CAQL query and a path expression. Since the path expression is generated and transmitted to the CMS before the actual CAQL queries it represents are submitted by the IE, the CMS must be able to keep track of the path expression element to which a given CAQL query corresponds. Path expression tracking is crucial if path expressions are to be of any use to the CMS. For example, consider the following excerpt from a path expression.

$$(...(d_1(X?,Y\hat{}),$$
$$[(\,d_2(Z\hat{},Y?),\, d_3(Z?)\,),$$
$$(\,d_4(U\hat{},Y?),\, d_5(U?)\,)\,]^{1}\,)^{<0,|X|>}\, ...\,)^{<0,1>}$$

where $<0,|X|>$ is a repetition count with a lower bound of zero and an upper bound equal to the cardinality of the bindings for the variable X obtained when the query corresponding to $d_1(X?,Y\hat{})$ is executed. The super-script of 1 after the alternation grouping is the selection term which indicates that at most one element of the grouping will be required for any given repetition of the grouping (i.e., the elements of the grouping are mutually exclusive). For the path expression given above, the following are some valid sequences of CAQL queries (only the relation names are shown).

d_1
d_1, d_2, d_3
$d_1, d_4, d_1, d_2, d_3, d_1, ...$
$d_1, d_2, d_3, d_1, d_4, d_5, ...$

After the CMS receives the CAQL query d_1 it can predict that the next query (if any) will involve either d_2 or d_4. Assume that the next query involves d_2. Now the CMS can predict that the next query will involve d_3 or d_1 (since d_1 could be repeated). Further, it can predict that if the next query involves d_3 then the query after that (if any) will involve d_1 (since the alternation is mutually exclusive).

Thus, d_1 will be required for one of the next two queries. If the CMS needs to replace some cache element it is clear that d_1 is not the best candidate.

5. Cache Management System

The Cache Management System (CMS) facilitates interactions between the IE and the remote DBMS. It achieves efficiency by alleviating the differences between the two systems and by allowing each to perform those operations for which it is best suited. Its distinguishing features include the following:

(a) A query language (CAQL) which is a superset of conventional, relational query languages such as SQL and the support of all CAQL operations by the CMS.

(b) The use of subsumption and query decomposition to identify subqueries that can be processed using previously cached data.

(c) The use of advice from the IE to improve overall performance *via* improved cache management including prefetching, indexing, and query generalization.

(d) Support for both eager and lazy evaluation of queries.

(e) Support for parallel execution of subqueries on both the CMS and the remote DBMS.

(f) Interfaces for efficient data transfer which use stream processing, buffering, and where possible, pipelining.

While a detailed description of all features of the CMS is not possible due to space limitation, we give brief descriptions of the above features with a relatively more detailed treatment of items (b) and (c). In Section 5.3, we demonstrate some of the uses of advice for an IE using a Prolog style of processing, although the CMS has also been designed to support IEs that perform more compilation.

As mentioned in Section 3, a session of IE – CMS interactions consists of a set of advice followed by a sequence of CAQL queries. Furthermore, queries submitted by the IE are not the only queries processed by the CMS. The CMS may generate additional queries as a result of using the advice it has. In the following, the term "IE-query" means a query given by the IE while the term "query" is used for an IE-query or a query generated by the CMS.

The organization of the CMS is shown in Figure 5. The Query Planner/Optimizer (QPO) plans execution of each query given to it by the parser or internally generated due to the advice and generates a program consisting of a partially ordered set of subqueries where each subquery is designated for execution by either the Cache Manager or by the remote DBMS. The Advice Manager interacts with the QPO to assist in query planning and optimization and with the Cache Manager to assist in caching and replacement decisions. The Execution Monitor coordinates the execution of the subqueries according to the order specified by the QPO. Subqueries to the remote DBMS can be executed in parallel with the subqueries to the Cache Manager. Queries to the remote DBMS are translated from CAQL to the DML of the DBMS by a DBMS specific translator in the Remote DBMS Interface (RDI). The RDI interacts with the remote DBMS via a standard communication protocol, and buffers the data returned by the DBMS prior to passing buffer control to the Cache Manager for caching that data. The Cache Manager manages the cache, the cache model, and (a copy of) the remote database schema. The cache consists of cache elements. A cache element is a relation defined by a CAQL expression that may contain base relations, simple or nested views over the base relations, and evaluable functions involving the cached relations. The cache model represents the state and statistical information about the cache (i.e., it is a schema of the cache). The Query Processor, an integral component of the Cache Manager, performs the actual DBMS-like operations (i.e., joins, selects, aggregation, indexing, etc.) on the cache elements.

A CAQL query is a well formed formula in quantified, first-order predicate calculus. In CAQL, predicate names are symbols which are mapped through a dictionary into: (a) explicit relations and views stored in the remote database or the cache; (b) comparison relations (e.g., less than); and/or (c) relations derived by computation over some of the arguments. CAQL supports arithmetic operators, logical connectives (AND, OR, NOT), special second-order predicates (BAGOF, SETOF, AGG, etc.), and quantifiers (ALL, EXISTS, ANY, THE).

5.1. Data Representation: Extensions and Generators

The CMS represents a relation as either the full extension of the relation or as a *generator* which produces a single tuple on demand. We use the term *eager evaluation* to describe the process of producing the extension of a relation, and the term *lazy evaluation* to describe an evaluation using a generator. This capability of representing relations as generators is one of the differentiators between the CMS and conventional DBMSs. Lazy evaluation has been supported internally in some DBMSs, but has not been available externally so that it can be used by an application or a user. Of course, lazy

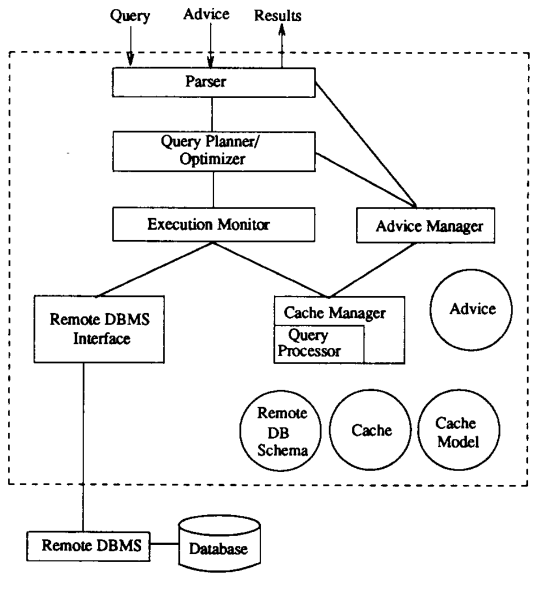

Figure 5: Organization of the CMS

evaluation can only be supported by the CMS when all required data is in the cache.

The CMS decides between the extension or generator as the representation for a given CAQL expression depending on (a) the kind of use that is to be made of the relations, (b) whether the cost of constructing the extension is reasonable, and (c) whether cache space is available for storage of the extension. For example, lazy evaluation is advantageous when the IE may require only a small subset of the relation and the cost of producing that subset is significantly less than the cost of producing the full extension or when the relation, or any relation from which it is derived, would exceed the capacity of the cache. In addition, the IE helps the CMS decide on the preferred representation by providing advice indicating the argument binding pattern for each cache element (see Subsection 5.3.3).

5.2. Co-existing, alternative representations of the same relation

The CMS frequently maintains co-existing, alternative representations of the same relation, another major differentiator of the CMS from a conventional DBMS. Consider, for example, the case where alternative sortings are required -- or the case where a relation will be used in different contexts, one where it serves as a producer of values in sequence (and can thus best be represented as a generator) and another where it needs repeatedly to be searched for particular values (and can thus best be represented as an appropriately indexed extension).

Advice from the IE informs the CMS of different uses to be made of the same relation, with each of the uses uniquely named. In many cases, the CMS is able to use a single instance of the relation in the cache (either as a generator or as an extension) to represent more than one of these uses that are distinguished from each other with separate names. The CMS makes the decision whether common representation for separate uses is feasible and efficient.

5.3. Query Planning and Optimization

The following techniques are employed by the CMS to improve the efficiency of data access and query processing:

Caching of relevant data: This reduces communication costs and other overhead of accessing the remote DBMS. The relevant data may be cached by prefetching data or by generalizing a CAQL query as indicated by advice. With generalization, the CMS retrieves more data from the DBMS (and caches it) than is required for a given CAQL query. The assumption is that later queries can be solved using the additional data and thus reduce the number of separate DBMS requests.

Result caching: This eliminates the cost of recomputing repeated CAQL queries.

Subsumption and operations on cached data: Subsumption followed by local query execution on relevant cached data is used to solve a query, when no result for the query exists, thereby improving the reusability of the cached data. Performing some operations in the cache will be more efficient in some cases than performing them in the remote DBMS or the IE.

Additional Operations: Performing operations that are not supported by the remote DBMS improves performance in those cases where performing the operation in the CMS is done more efficiently than performing it in the IE.

Parallel execution between the CMS and the DBMS: Parallel execution of subqueries on both the cache and on the remote DBMS improves efficiency.

A detailed discussion of the query processing and optimization task is beyond the scope of this paper. Here we will only discuss some uses of the advice, mostly through examples.

Operations of the QPO can be divided into three steps. The first step is to determine the query to be evaluated. The second step is to identify relevant cache elements that can possibly be used in processing all or a part of the query. The third step is to generate a plan that consists of a partially ordered set of subqueries to be evaluated by the Cache Manager and the remote DBMS.

5.3.1. Step 1: Determine the Query to be Evaluated

An IE-query is an instance of one of the view specifications with constant bindings. The CMS checks against the advice and decides whether it wants to process the given IE-query or a more general query that amounts to prefetching additional data follow by operations to produce the result of the IE-query. This requires using the subsumption to check if the IE-query or its parts can be subsumed by any other view specification or its parts.

Now let us continue discussion of example 1 of section 4. As indicated earlier, the possible IE-queries are $d_1(Y)$, $d_2(X,c)$, and $d_3(X,c)$, where c is any constant. If a part of an IE-query as defined in the view specification is subsumed by another view specification, then the query to be evaluated may be modified to be of a more general form. For example, the query $d_1(X,Y)$ will be translated into query $b_1(c_1,Y)$. However, there is $b_1(X,Y)$ in the definition of d_3 that subsumes $b_1(c,Y)$. Hence the CMS may consider evaluating $b_1(X,Y)$ that is a generalization of the query $b_1(c_1,Y)$. The algorithm used to find subsumption among view specifications is the same as that discussed in step 2 below. The decision of whether to generalize will depend on evaluating the cost of alternatives as discussed in step 3.

The path expression may be used by the CMS in deciding to prefetch data. The sequence grouping (defined in Subsection 4.2.2) in a path expression indicates that all items in that group are likely to be evaluated when the first item is evaluated. For example, assuming that all solutions are sought for the query $k_1(X,Y)$ and that the evaluation of $d_1(Y)$ yields $|Y| > 0$, the CMS may decide processing $d_3(X,c)$ soon after it processes $d_2(X,c)$ and before it actually receives $d_3(X,c)$ from the IE.

5.3.2. Step 2: Determine Relevant Cache Elements

Let Q be a query expression to be processed by the CMS. A subquery Q_c of Q is any conjuctive portion of Q. The cache is a set of n cache elements, E_i's ($i = 1,\dots,n$). Both Q and an E_i can involve joins, projections, and selections among multiple relations. A cache element is a derived view relation, either stored in an extensional form (and hence a materialized view), or in a generator form (which can be used for lazy evaluation).

The cache model contains information on the cache elements. It is a relation of type $(E\_id_i, E\_def_i, ...)$, where E_id_i is the identifier of E_i, E_def_i is its definition (usually a view specification), etc.

Given a cache query Q, we need to find out which cache elements can be used to derive *all or a component of* Q. In other words, the problem is to find all Q_c of Q, such that Q_c is derivable from an E_i (i.e., there exists an $E_i \supset Q_c$, where "$\supset$" stands for "subsumes" or "can be used to derive"). The set of all E_is that can be used for deriving any Q_c of Q is a set of relevant elements, $R\{E_i\}$ of Q. Not all relevant E_i's may be used in solving the Q. The subqueries that are not derived from cached elements can be derived from the database. Such a reuse of cached data clearly extends the efforts to reuse cached data in earlier AI/DB integration efforts. In [SELL87] and [IOAN88], the cached results must exactly match the query. In [CERI86], cached elements contain only single relations. We limit Q and E_is to logic expressions equivalent to *PSJ* expressions (as in [LARS85]) which make the problem more constrained than the more general implication problem [SUN89]). For an evaluable function, we require the exact match. Solving derivability for each possible component of Q (and for a $|Q|=n$, there are $n(n+1)/2$ components) may not be efficient. A sketch of our approach, which is discussed in more detail in [SHET88], is as follows.

1. Consider subqueries of single predicates and the cache elements that have the same predicate in their definitions. An index of type (*predicate name, cache element*) can expedite this process. If the predicate in the cache element can subsume the predicate in the subquery, consider the cache element further. Check for subsumption requires matching the predicate in the subquery with the predicate in the cache element. This matching is like a unification in a single direction; a constant in the predicate in the subquery can match with the same constant or a variable at the corresponding position in the predicate in the cache element, but a variable can only match with a variable. Consider $Q_{c1} = b_{21}(X,2)$, and $E_1 = b_{21}(X,Y)$ & $b_{22}(Y,Z)$, $E_2 = b_{21}(3,Y)$ and $E_3 = b_{21}(X,2)$ & $b_{23}(2,Z)$. Here E_1 and E_3 will be considered further with the "unifier"s $(\_, Y=2)$ and $(\_,\_)$, respectively, for $b_{21}(X,2)$.
2. Consider the predicates to the left and the right of the predicate considered in step 1. If the query does not have the same respective predicates that are also subsumed by the predicates in the cache element, then the cache element is more restricted, and cannot be used to derive the query component. All remaining subqueries and cache elements may be considered further. For example, for $Q_{1a} = b_{21}(X,2)$ & $b_{22}(2,Y)$, $Q_{1b} = b_{23}(2,3)$ & $b_{21}(X,2)$, $Q_{1c} = b_{21}(2,Y)$ & $b_{23}(Y,Z)$, and E_3 as above, E_3 will be considered only for Q_{1b}.

Let us further consider example 1 discussed in Subsections 4.2.2 and 5.3.1. Assume the cache to contain following elements:

E_{11}: $b_2(X,c_1)$ & $b_3(Y,c_2,c_6)$
E_{12}: $b_3(X,c_2,Y)$
E_{13}: $b_3(X,Y,Z)$

In processing a query $d_2(X,c_6) = b_2(X,Z)$ & $b_3(Z,c_2,c_6)$, the CMS will identify that either E_{12} or E_{13} can be used to compute the $b_3(X,c_2,Y)$ part of the query.

5.3.3. Step 3: Generate Plan

Plan generation involves dividing the CAQL query into a partially ordered set of subqueries where each subquery can be executed either by the Cache Manager or by the remote DBMS.

Step 2 identifies all relevant cache elements that may be used to evaluate a part of the query. When multiple cache elements overlap, i.e., can be used to evaluate the same part of the query, than the most appropriate element has to be chosen. For example consider a query $b_1(X,Y)$ & $b_2(Y,c_1)$ and the following cache elements:

E_{101}: $b_1(X,Y)$
E_{102}: $b_2(X,c_1)$
E_{103}: $b_1(X,Y)$ & $b_2(Y,Z)$

Here the CMS may find the doing the join between E_{101} and E_{102} to solve a subquery may be more expensive than performing a selection on E_{103} $(Z=c_1)$ and hence remove E_{101} and E_{102} from further consideration. The remaining elements are further considered in query planning and optimization.

The information on the producer and consumer variables provided in the advice is one of the considerations used by the CMS to determine whether to evaluate a query eagerly and store the extension or to evaluate the query lazily by storing a generator expression for evaluating it on demand. A general guideline is as follows:

- For an expression of kind $d(X^\wedge,Y^\wedge)$, evaluate lazily if all required data is in the cache.
- For an expression of kind $d(X?,Y^\wedge)$ (or $d(X?,Y?)$), evaluate eagerly and create an index on X (X and Y) unless the result is expected to be too large and may not fit in the cache.

At this time, the QPO has information on all the relevant cache elements and how the query is to be evaluated (lazily or eagerly). Assuming the eager evaluation, it uses the cost functions to prune the search space and arrive at the plan to be followed. The plan specifies parallel executions of the subqueries for the remote DBMS and the CMS whenever possible.

Query optimization can be very complicated because the problem search space, which includes all the alternatives for processing, is very large. In addition to the the usual optimization issues in relational DBMS optimization, such as ordering of operations and indexing the relations on join attributes, there are several complicating factors including (a) part of the remote database is duplicated in the cache; (b) the costs of performing an operation in the cache and in the remote DBMS are different; (c) the cost of communicating with remote DBMS is significant; (d) the DBMS and the CMS do not support the same set of operations (the remote DBMS does not support all CAQL operations, but the CMS does); and (e) there are two methods of evaluating some operations, i.e., eager evaluation and lazy evaluation. Further, advice from the IE provides global information about the AI query which can be used by the CMS in planning but it also increases the size of the planning search space. For example, advice may be used to generalize a query (retrieve more data from the remote database than is necessary for the given

CAQL query assuming that the additional data will be useful for subsequent queries), but characterizing the advantage of doing so is difficult. The end result is an optimization problem that is far more complicated than for a distributed DBMS and naturally involves some significant overhead. Determining which optimizations result in savings that outweigh the associated overhead is one of the most important long term objectives of this project.

In continuing to discuss processing $d_2(X, c_6)$ in example 1 (subsections 4.2.2 and 5.3.2) with the cache containing two relevant elements, E_{12} and E_{13}, the QPO will decide that using E_{12} is preferable. Now the QPO will evaluate the cost of following alternate plans. (a) Get $b_2(X, Y)$ from the DBMS, store it in the cache as say E_{14} (this involves considering the estimated advantage due to future cache hits on $b_2(X, Y)$), index E_{12} on the third attribute (because it was annotated as a consumer variable in the view specifications), perform $\sigma_{Y=c_\bullet} E_{12}$, and join the latter with E_{14}. (b) Export $b_2(X, Y)$ & $b_3(Z, c_2, c_6)$ to the DBMS. In either of the two cases, the result of evaluating $d_2(c_2, Y)$ will be stored in the cache and also returned to the IE using a stream.

5.4. The Cache Manager

The primary responsibilities of the Cache Manager include (a) maintaining the cache as well as storing and replacing cache elements (using an LRU scheme which may be modified due to advise); (b) executing queries on cached data in the working memory; (c) keeping track of resources consumed by the cached data; and (d) maintaining sufficient historical meta-data to support cache replacement and accumulate performance measurement statistics.

The Query Processor provides the Cache Manager with the capability to perform relational operations and aggregation on the data stored in the cache. It uses hash indices when available to speed up joins and some selections.

5.5. Data Transfer Interface Characteristics

Interfaces in *BrAID* have two aspects: the language for sending commands and queries and the data transfer scheme. The language aspect of the IE–CMS interface consists of CAQL and advice as previously described. The CMS–DBMS language interface is given by the DML of the remote DBMS. The Remote DBMS Interface component of the CMS performs the required translation and it provides the data transfer interface for *BrAID* with the remote DBMS.

In *Tangram* [PARK89], stream processing has been used to solve impedance mismatch between a Prolog and a DBMS. Stream processing buffers the results produced by the DBMS and passes results, one at a time, as they are requested by Prolog's inference engine. While such an interface supports a form of lazy evaluation (i.e., a stream will produce a tuple on demand), a conventional DBMS may perform more evaluation (either produce the entire result or enough to fill a buffer) than required by the inference engine. This problem, however, is mitigated in *BrAID* by complementing stream processing with lazy evaluation that produces a single solution on demand whenever possible (i.e., when a query can be solved using only cached data).

The CMS's interface to the remote DBMS provides buffers for the data returned by the DBMS. The interface also allows pipelining if the DBMS supports it. In that case, the DBMS starts returning the data before the complete result to the DBMS query has been processed. Buffering allows the CMS to continue processing operations by the Cache Manager at the same time data is being received from the remote DBMS.

6. Conclusions

We have described the architecture of *BrAID*, a system for the efficient integration of AI systems with a conventional DBMS. A key design criterion for *BrAID* is that the DBMS must be kept independent so that existing databases can be used without modification. To that end, the system employs a powerful, cache-based interface to complement deficiencies in the DBMS. Novel features of *BrAID* discussed in this paper are:

- *Advice*, obtained by pre-analysis of the AI application, that provides the cache based interface with information crucial to maximizing the efficiency of database accesses.
- The *caching of views* which can be defined in CAQL and are not limited to extensions of single relations or joins, and serve as the basis of *co-existing, alternative representations* of the same extensional data. Support for eager as well as lazy evaluation by the CMS.
- More powerful *subsumption* to compare a query and its components with cached views (including selects, projects, and joins) to allow a higher degree of data reuse and thus minimize DBMS requests.
- Use of advice by the CMS to increase efficiency in remote DBMS access through the selective use of prefetching, query generalization, attribute indexing, lazy evaluation and replacement.

The cache based architecture for AI/DB systems is of particular relevance to a large number of organizations that have substantial investments in the databases managed by the current generation of DBMSs. Many of the above features, however, can be gainfully employed in other approaches to AI/DB integration discussed in Section 1. The designers of future DBMSs may also find these to be beneficial for supporting a broader range of applications.

An experimental prototype with most of *BrAID*'s features has been implemented [SHET89]. The IE and the CMS are written in Xerox Common Lisp and reside on a Xerox workstation connected *via* Ethernet either to an INGRES on a Sun workstation or to a Britton-Lee IDM-500 database machine. Although preliminary performance results are encouraging [SHET89], more evaluations are needed to understand several important trade-offs related to the novel features of *BrAID*. We are also interested in investigating the utility of using the CMS for interfacing DBMSs with systems other than logic based inference engines.

Acknowledgements: *BrAID* is a result of a team effort. Other team members include Darrel Van Buer, Larry Travis, and Don McKay. Margaret Raid-Miller, Steve Russell, Son Dao, Rich Fritzson and Margaret Heineman also contributed to this project at various stages.

REFERENCES

[ABAR86] R. Abarbanel and M. Williams, "A Relational Representation for Knowledge Bases," Technical Report, Intellicorp, Mountain View, CA, April 1986.

[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.

[CERI86] S. Ceri, G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," *Proc. of the 1st Intl. Conf. on Expert Database Systems*, South Carolina, April 86.

[CHAN84] C. Chang and A. Walker, "PROSQL: A Prolog Programming Interface with SQL/DS," *Proc. of the 1st Intl. Workshop on Expert Database Systems*, Kiawah Island, South Carolina, October 1984.

[CHIM87] D. A. 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.

[IOAN88] Ioannidis, Y., J. Chen, M. Friedman, and M. Tsangaris, "BERMUDA -- An Architectural Perspective on Interfacing Prolog to a Database Machine," *Proc. of the 2nd International Conference on Expert Database Systems*, April 1988.

[IOAN90] Ioannidis, Y., and M. Tsangaris, "The Design, Implementation, and Performance Evaluation of BERMUDA," Computer Sciences Technical Report #973, University of Wisconsin- Madison, October 1990.

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

[KELL81] Kellogg, C. 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] Kellogg, C., A. O'Hare, and L. Travis, "Optimizing the Rule/Data Interface in a Knowledge Management System", *Proc. of the 12th Intl. Conf. on Very Large Databases*, Kyoto, Japan, 1986.

[KOGA86] Kogan D., "The FDE Internal DBMS: A User's Manual and Programmer's Guide," Technical Memo (unpublished), Systems Development Corporation, December 1986.

[LARS85] A. Larson and H. Yang, "Computing Queries from Derived Relations," *Proc. of the 11th VLDB*, Stockholm, August 1985.

[MARQ84] Marque-Pacheu, G., 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.

MCKA90] McKay, D., T. Finin, and A. O'Hare, "The Intelligent Database Interface: Integrating AI and Database Systems", *Proc. of the 7th National Conf. on Artificial Intelligence (AAAI-90)*, Boston MA, July-August 1990.

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

[OHAR89a] A. O'Hare and L. E. Travis, "The KMS Inference Engine&gml. Rationale and Design Objectives", Technical Report PRC-LBS-8919, Unisys -- Paoli Research Center, January 1989.

[OHAR89b] A. O'Hare and A. Sheth, "The Interpreted–Compiled Range of AI/DB Systems," SIGMOD RECORD, March 1989.

[PARK89] D. Stott Parker, R. Muntz, and H. Chau, "The Tangram Stream Processing System," *Proc. of the 5th Intl. Conf. on Data Engineering*, Los Angeles, CA, February 1989.

[SELL87] T. Sellis "Intelligent Caching and Indexing Techniques for Relational Database Systems," Technical Report UMIACS-TR-87-6, University of Maryland, 1987.

[SHET88] A. Sheth, D. Van Buer, S. Russell, and S. Dao, "Cache Management System: Preliminary Design and Evaluation Criteria," Unisys Technical Report TM-8484/000/00 (Contact: A. Sheth), October 1988.

[SHET89] A. Sheth and A. O'Hare, "The Architecture of BrAID: A System for Efficient AI/DB Integration," Technical Memo TM-STS-015544, Bellcore, November 1989.

[STON87] M. Stonebraker, E. Hanson and S. Potamianos, "A Rule Manager for Relational Database Systems," in *The Postgres Papers*, M. Stonebraker and L. Rowe (eds), Memo UCM/ERL M86/85, Univ. of California, Berkeley, 1987.

[SUN89] X. Sun, N. Kamel, and L. Ni, "Solving Implication Problems in Database Applications," *Proc. of the SIGMOD*, June 1989.

[VANB85] D. Van Buer, D. McKay, D. Kogan, L. Hirschman, M. Heineman, and L. Travis, "The Flexible Deductive Engine: An Environment for Prototyping Knowledge Based Systems," *Proc. of the 9th International Joint Conference on Artificial Intelligence*, Los Angeles CA, August 1985.