The Intelligent Database Interface: Integrating AI and Database Systems

Donald P. McKay and **Timothy W. Finin** and **Anthony O'Hare***
Unisys Center for Advanced Information Technology
Paoli, Pennsylvania
mckay@prc.unisys.com and finin@prc.unisys.com

Abstract

The *Intelligent Database Interface* (IDI) is a cache-based interface that is designed to provide Artificial Intelligence systems with efficient access to one or more databases on one or more remote database management systems (DBMSs). It can be used to interface with a wide variety of different DBMSs with little or no modification since SQL is used to communicate with remote DBMSs and the implementation of the IDI provides a high degree of portability. The query language of the IDI is a restricted subset of function-free Horn clauses which is translated into SQL. Results from the IDI are returned one tuple at a time and the IDI manages a cache of result relations to improve efficiency. The IDI is one of the key components of the *Intelligent System Server* (ISS) knowledge representation and reasoning system and is also being used to provide database services for the Unisys spoken language systems program.

Introduction

The *Intelligent Database Interface* (IDI) is a portable, cache-based interface designed to provide artificial intelligence systems in general and expert systems in particular with efficient access to one or more databases on one or more remote database management systems (DBMS) which support SQL [Chamberlin, et. al., 1976]. The query language of the IDI is the Intelligent Database Interface Language (IDIL) [O'Hare, 1989] and is based on a restricted subset of function-free Horn clauses where the head of a clause represents the target list (i.e., the form of the result relation) and the body is a conjunction of literals which denote database relations or operations on the relations and/or their attributes (e.g., negation, aggregation, and arithmetic operations).

The IDI is one of the key components of the *Intelligent System Server* (ISS) [Finin, et. al., 1989] which is based on *Protem* [Fritzson and Finin, 1988] and provides a combined logic-based and frame-based knowledge representation system and supports forward-chaining, backward-chaining, and truth maintenance. The IDI was designed to be compatible with the logic-based knowledge representation scheme of the ISS and its tuple-at-a-time inference mechanisms. The IDI has also been used to implement a query server supporting a database used for an *Air Travel Information System* which is accessed by a spoken language system implemented in Prolog [Dahl, et. al., 1990].

In addition to providing efficient access to remote DBMSs, the IDI offers several other distinct advantages. It can be used to interface with a wide variety of different DBMSs with little or no modification since SQL is used to communicate with the remote DBMS. Also, several connections to the same or different DBMSs can exist simultaneously and can be kept active across any number of queries because connections to remote DBMSs are abstract objects that are managed as resources by the IDI. Finally, accessing schema information is handled automatically by the IDI, i.e., the application is not required to maintain up-to-date schema information for the IDI. This significantly reduces the potential for errors introduced by stale schema information or by hand entered data.

The IDI can be viewed as a stand-alone DBMS interface which accepts queries in the form of IDIL clauses and returns the result relation as a set of tuples (i.e., a list of Lisp atoms and/or strings). IDIL queries are translated into SQL and sent to the appropriate DBMS for execution. The results from the DBMS are then transformed by the IDI into tuples of Lisp objects. Although the IDI was not designed to be used directly by a user, the following descriptions will be couched in terms of using the IDI as a stand-alone system so that we may avoid complicating our discussions with the details of an AI system such as the ISS.

The design of the IDI was heavily influenced by previous research in the area of AI/DB integration [Kellog, et. al., 1986, O'Hare, 1987, O'Hare and Travis, 1989, O'Hare and Sheth, 1989]. One of the more significant design criteria that this lead to is the support of non-trivial queries in IDIL. That is, to allow for queries involving more than just a single database relation. This capability allows the AI system to off-load computations that are more efficiently processed by the DBMS instead of the AI system (e.g., join operations). In many cases, this also has the effect of reducing the size of data set that is returned by the DBMS.

*current address: IBM, Research Triangle Park, North Carolina

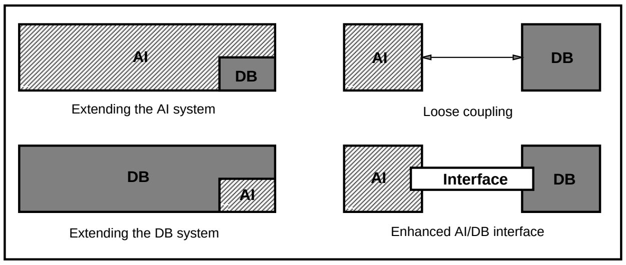

Figure 1: **Of the four alternative approaches to AI/DB integration, the** *Intelligent Database Interface* **is an example of an enhanced AI/DB interface.**

While the IDI is to some small degree system dependent, it does offer a high degree of portability because it is implemented in Common Lisp, communicates with remote DBMSs using SQL and standard UNIX pipes, and represents IDIL queries and their results as Common Lisp objects.

In the following sections we present a brief overview of the area of AI/DB integration which represents a large part of the motivation for the IDI, a discussion of some of the more significant features of the IDI, the organization and major components of the IDI, and finally an example of how the IDI is being used in two applications.

AI/DB Integration

The integration of AI and DBMS technologies promises to play a significant role in shaping the future of computing. As noted in [Brodie, 1988], 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.

While both DBMS and AI systems, particularly expert systems, represent well established technologies, research and development in the area of AI/DB integration is comparatively new. The motivations driving the integration of these two 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 IDI was also motivated by the desire to preserve the substantial investment represented by most existing databases. To that end, a key design criterion for IDI was that it support the use of existing DBMSs as independent system components. As illustrated in Figure 1 and described below, several general approaches to AI/DB integration have been investigated and reported in the literature (e.g., [Bocca, 1986, Chakravarthy, et. al., 1982, Chang, 1978, Chang and Walker, 1984, Jarke, et. al., 1984, Li, 1984, Minker, 1978, Morris, 1988, Naish and Thom, 1983, Reiter, 1978, Van Buer, et. al., 1985]).

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. In general, such systems do not incorporate full DBMS technology. Rather, the emphasis is on the AI system and the DBMS capabilities are added in an *ad hoc* and limited manner, e.g., [Ceri, et. al., 1986] implements only the data access layer. Alternatively, a new generation knowledge-based system such as LDL [Chimenti, et. al., 1987] may be constructed. In either case, this approach effectively involves "re-inventing" some or all of DBMS technology. While such systems typically provide sophisticated tools and environments for the development of applications such as expert systems, they can not readily make use of existing databases. Thus, the development of AI applications which must access existing databases will be exceedingly difficult if not impossible (e.g., when the database is routinely accessed and updated *via* more traditional kinds of applications).

Extending the DBMS System: This approach extends a DBMS to provide knowledge representation and reasoning capabilities, e.g., POSTGRES [Stonebreaker, et. al., 1987]. Here, the DBMS capabilities are the central concern and the AI capabilities are added in an *ad hoc* manner. The knowledge representation and reasoning capabilities are generally quite limited and they lack the sophisticated tools and environments of most AI systems. Such systems 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. In some sense, this is the opposite of the previous approach.

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 [Abarbanel and Williams, 1986]. While this approach has the distinct advantage of integrating existing AI systems and existing DBMSs, 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 representing data (e.g., relational data models *vs.* frames) is generally left to the application developer or knowledge engineer with only minimal support from the AI/DB interface.

Enhanced AI/DB Interface: 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 AI and 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. For example, the BERMUDA system [Ioannidis, et. al., 1988] 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. The BrAID system [O'Hare and Sheth, 1989] is similar except that it supports more general caching and pre-analysis capabilities and allows for experimentation with different inference strategies.

The IDI is an interface that can be used to facilitate the development of AI/DB systems using this last approach. That is, the IDI is cache-based interface to DBMSs and is designed to be easily integrated into various types of AI systems. The design of the IDI also allows it to be used as an interface between DBMSs and other types of applications such as database browsers and general query processors.

Design Features

The IDI supports several features which simplify its use in an AI system. These include

- Connections to a DBMS are managed transparently so that there can be multiple active queries to the same database using a single open connection.
- Connections to a given database are opened upon demand, i.e. at first use instead of requiring an explicit database open request.
- Database schema information is loaded from the database either when the database is opened or when queries require schema information based upon user declarations.
- The query interface is a logic-based language but uses user supplied functions to declare and recognize logic variables.
- Results of queries to a DBMS are cached, improving the overall performance system and the cache is accessed transparently by a query manager.

All but the last of the features are described in this section. The cache system and initial performance results are described in subsequent sections.

Making Connections

As suggested above, there are numerous approaches to interfacing an AI system with existing DBMSs. However, the basic alternatives involve balancing the costs of creating the connection to the DBMS and of processing the result relations from a DBMS query. Deciding which alternative is the best requires knowledge about the typical behavior of the AI system as well as other, more obvious factors, such as communication overhead (cf. [O'Hare and Sheth, 1989]). Consider the following two modes of interaction between an AI system and a DBMS:

- The AI system generates a few DBMS queries that tend to yield very large results and the AI system uses only a fraction of the result.
- The AI system generates many DBMS queries that tend to yield, on average, small results and the AI system uses most or all of the result.

In the first case, it would be best to avoid the cost of processing the entire result by using demand-driven techniques to produce only one tuple at a time from the result stream of the DBMS. However, this requires that separate connections be created for each DBMS query. Thus the overhead of creating such connections must be less than the cost of processing the entire result relation.

In the second case, it would be best to avoid the cost of creating numerous connections to the DBMS by using a single connection for multiple queries. However, this requires that the entire result of each query be processed so that successive queries can be run using the same connection. The cost of processing DBMS results (i.e., reading the entire result stream and storing it locally) must be less than the cost of creating a new connection for each DBMS query.

For most systems, it seems reasonable to assume that the total cost for creating a new DBMS connection will be relatively high. Thus, using the same connection for different DBMS queries would result in a net savings. While specific break-even points could be estimated, it is not clear one need go that far since there are other reasons for minimizing the number of DBMS connections that are open at the same time. Foremost among these is the limit that most operating systems have on the number of streams that can be open simultaneously. This can severely limit the number of DBMS connections that we can have at one time. If one is also interested in allowing connections to different databases, on either the same or a different DBMS, then it is important to minimize the number of open connections for a single database.

Yet another consideration is the use of caching for DBMS results. That is, if DBMS results can be cached locally by the AI system or an agent serving it then all of the DBMS results will probably be processed by the caching mechanism. Thus, the first alternative (where it is assumed that the DBMS results will not, in general, be totally consumed) is no longer applicable.

In light of these constraints and requirements, it seems best to minimize the number of DBMS connections that can be open simultaneously. Briefly, the approach taken in the IDI is to open a connection when a DBMS query is encountered against a database for which no connection exists and process the result stream one tuple at a time until and unless another DBMS query on the same database is encountered. At that point, the new query is sent to the DBMS, the remainder of the result stream for the previous query is consumed and stored locally, and then the new result stream is processed one tuple at a time as before.

Automating Access to Schema Information

One of the key features of the IDI is the automatic management of database schema information. The user or application program is not required to provide any schema information for those database relations that are accessed *via* IDIL queries. The IDI assumes the responsibility for obtaining the relevant schema information from the appropriate DBMS. This provides several significant advantages over interfaces which rely on the user to provide schema information. Most importantly, the schema information will necessarily be consistent with that stored in the DBMS and thus any errors introduced by hand-coding the schema information are eliminated. The only exception to this occurs when the schema on the DBMS is modified after the IDI has accessed it since the IDI caches the schema information and thus maintains a private copy of it. While this *stale data* problem exists for any system which maintains a separate copy of the schema information, the IDI provides a simple mechanism for forcing the schema information to be updated. In addition, this approach greatly facilitates the implementation of database browsers since users need not know the names or structure of relations stored in a particular database.

Logical Glue

Another significant feature of the IDI is the relative ease with which it can be integrated with different AI systems. Aside from the use of Common Lisp as the implementation language for the IDI, this is achieved by employing a logic-based language as the query language for the IDI. The language, IDIL, may be used as a totally independent query language or, more importantly, it may be more closely integrated with the knowledge representation language of a logic-based AI system. In the later case, the key is to allow the IDI to share the same definition of a logic variable as the host AI system. This is accomplished be simply redefining a small set of functions within the IDI which are used to recognize and create instances of logic variables.

The IDIL [1] query language is a restricted subset of function-free Horn clauses where the head of a clause represents the target list (i.e., the form of the result relation) and the body is a conjunction of literals which denote database relations or operations on the relations and/or their attributes (e.g., negation, aggregation, and arithmetic operations). Figure 2 shows some example queries.

Get supplier names for suppliers who do not supply part p2.

```
((ans _Sname)
 <-
 (supplier _Sno _Sname _Status _City)
 (not (supplier_part _Sno "p2" _Qty)))
```

Get supplier names and quantity supplied for suppliers that supply more than 300 units of part p2.

```
((ans _Sname _Qty)
 <-
 (supplier _Sno _Sname _Status _City)
 (supplier_part _Sno "p2" _Qty)
 (> _Qty 300))
```

Figure 2: **Two example IDIL queries using the "suppliers" database. Symbols beginning with a "_" character have been declared to be logic variables.**

IDI Organization

As Figure 3 illustrates, there are four main components which comprise the IDI — the *Schema Manager*, the *DBMS Connection Manager*, the *Query Manager*, and the *Cache Manager*. There are three principal types of inputs or requests to the IDI: (a) a database declaration; (b) an IDIL query and subsequent retrieval requests against the result of an IDIL query; and (c) advice to the Cache Manager.. Database declarations convey simple information about a given database, e.g., the type of the DBMS on which the database resides and the host machine for the DBMS. For each IDIL query, the IDI returns a *generator* which can be used to retrieve the result relation of an IDIL query one tuple at a time. The IDI also supports other types of requests, e.g., access to schema information, which are described elsewhere [O'Hare, 1989].

The *Schema Manager* (SM) is responsible for managing the schema information for all declared databases and it supplies the Query Manager with schema information for individual database relations. This entails processing database declarations, accessing and storing schema information for declared databases, and managing relation name aliases which are used when two or more databases contain relations with identical names. Whenever a connection to a database is created, the SM automatically accesses the list of relation names that are contained within the database. This list is then cached for later access in the event that the connection is closed and re-opened at some later time. In this event the SM will only access the DBMS schema information if it is explicitly directed to do so, otherwise the cached list of relation names will be used.

The *DBMS Connection Manager* (DCM) manages all database connections to remote DBMSs. This includes processing requests to open and close database connections as well as performing all the low-level I/O operations associated with the connections. Within the IDI, each database has at most one active connection associated with it and each connection has zero or more query result streams or *generators* associated with it but only one generator may be active.

The *Query Manager* (QM) is responsible for processing IDIL queries and managing their results. IDIL queries are processed by translating them into SQL

[1] "IDIL" is pronounced as "idle" and should not be confused with "idyll".

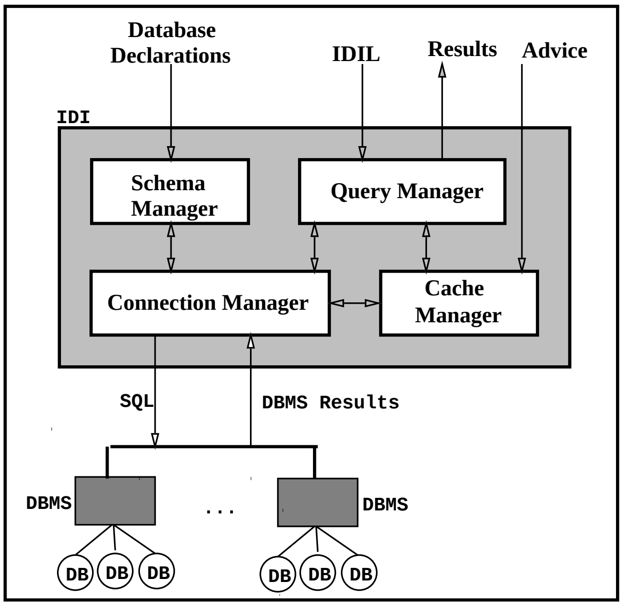

Figure 3: **The IDI includes four main components: the Schema Manager manages the schema information for all declared databases; the Connection Manager handles connections to remote DBMSs; the Query Manager is responsible for processing IDIL queries and their results; and the Cache Manager controls the cache of query results in accordance with the advice supplied by the application.**

which is then sent to the appropriate DBMS by the DCM. If the query is successfully executed by the DBMS then the QM returns a *generator* for the result relation. A generator is simply an abstract data type used to represent the result of an IDIL query. There are two basic type of operations which may be performed on a generator: (a) get the next tuple from the result relation and (b) terminate the generator (i.e., discard any remaining tuples). Generators are actually created and managed by the DCM since there is more than one possible representation for a result relation, e.g., it may be a result stream from a DBMS or a cache element. The QM merely passes generators to the DCM along with requests for the next tuple or termination.

The *Cache Manager* is responsible for managing the cache of query results. This includes identifying IDIL queries for which the results exist in the cache, caching query results, and replacing cache elements. In addition, our design allows the AI system to provide the cache manager with *advice* to help it decide how to manage its cache and make the following kinds of critical decisions:

- *pre-fetching* - which relations (and when) should be fetched in anticipation of needing them? This can yield a significant increase in speed since the database server is running as a separate process. This can also be used to advantage in an environment in which databases are accessed over a network in which links are unreliable – critical database relations can be accessed in advance to ensure their availability when needed.
- *results caching* - which query results should be saved in the cache? Both base and derived relations vary in their general utility. Some will definitely be worth caching since they are likely to be accessed soon and others not.
- *query generalization* - which queries can be usefully generalized before submitting them to the DBMS? Query generalization is a useful technique to reduce the number of queries which must be made against the database in many constraint satisfaction expert systems. It is also a general technique to handle expected subsequent queries after a "null answer" [Motro, 1986].
- *replacement* - which relations should be removed when the cache becomes full?

Additional kinds of advice and examples can be found in [O'Hare and Travis, 1989, O'Hare and Sheth, 1989].

As with any type of cache-based system, one of the more difficult design issues involves the problem of cache validation. That is, determining when to invalidate cache entries because of updates to the relevant data in the DBMS. Our current implementation does not attempt cache validation, which will be a focus of future research. This still leaves a large class of applications for which cache validation is not a problem. These includes access to databases that are write-protected and updated infrequently, such as the *Official Airline Guide* database, and databases that are static relative to the time scale of the AI application accessing them.

Moreover, this problem is common to any AI system which gets some of its data from an external source and stores in its knowledge base. Most current interfaces between AI systems and databases (e.g. KEE Connection [Intellicorp, 1987]) simply do not worry about this problem at all. Our approach attempts to minimize the AI system's copying of database data in two ways. First, by providing convenient and efficient access to the information in the database, the AI system developers will have less need to make a local copy of the data. Second, all of the database information that is copied (i.e., in the cache) is isolated in one place and can therefor be more easily "managed" – reducing the problem to "cache validation".

There are a number of approaches to the validation problem which vary in completeness and ease of implementation. Examples of possible components of a cache validation system include: only caching relations declared to be non-volatile, only caching data between scheduled DB updates, using heuristics (e.g., a decay curve) to estimate data validity, and implementing a "snoopy cache" which monitors the database transactions for updates which might invalidate the cached data.

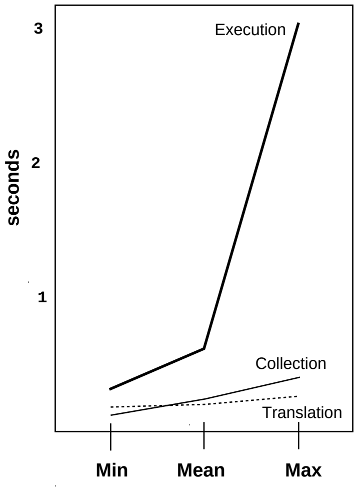

Figure 4: **The aggregate processing time is broken down in terms of the three main stages of processing:** *translation, execution,* **and** *collection.* **For each processing stage, the minimum, mean, and maximum processing times are shown.**

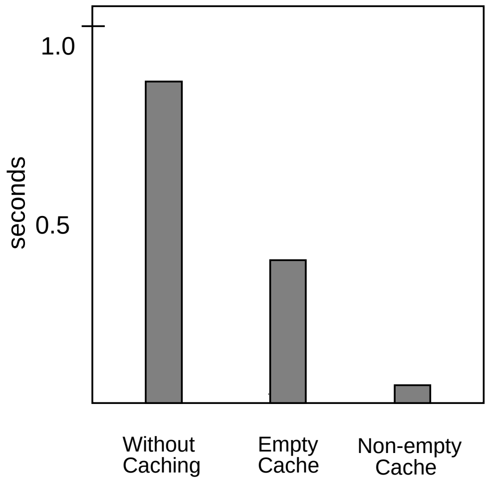

Figure 5: **Cache performance is measured for three cases: (a) the** *without caching* **or base-line case where caching was disabled, (b) the** *empty cache* **case where caching was enabled but the cache was cleared before each repetition of the test set, and (c) the** *non-empty cache* **case where the cache contained the results for all queries in the test set.**

Current Status

Performance

The IDI, as described here, has been implemented in Common Lisp and tested as a stand-alone query processor against two different databases running on RTI INGRES and is also being used as a query server for the Unisys spoken language project. The performance results obtained thus far are, at best, preliminary since the size of the test suite was comparatively small and the IDI is just now being integrated with an AI system. However, the results are encouraging and indicate the potential for efficient database access afforded by the IDI. The following summarize some of the more interesting of these performance results.

One test set of IDIL queries used consisted of 48 queries where there were 22 unique queries, i.e., each query was repeated at least once in the test set. The queries ranged from simple (i.e., only project and select operations were required) to complex (i.e., a four-way join with two aggregation operations as well as projects and selects). The size of the result relations varied from zero to 17 tuples. The statistics presented here are based on the mean processing times for 20 repetitions of the test set of queries.

Figure 4 shows a breakdown of the aggregate processing time in terms of the three main stages of processing: *translation* (i.e., the time to translate and IDIL query into SQL), *execution* (i.e., the elapsed time between sending the SQL query to the DBMS and obtaining the first tuple of the result relation), and *collection* (i.e., the time required to collect all the tuples in the result relation and convert them into internal form). For each processing stage, the minimum, mean, and maximum processing times are shown. The cache was disabled for these measurements so that a more accurate picture of the relative processing times for each stage could be established.

The differences in translation time reflect a dependence on the number of relations in the IDIL query. Similarly, the collection time is a function of the number of tuples in the result relation. In both cases, the processing times are significantly less than the execution time which is effected by the complexity of the SQL query, the communication overhead, and the load on the remote DBMS host (since only elapsed time was recorded).

Figure 5 indicates the effects of result caching on performance. The results represent the mean processing times (in seconds) for all queries. Three different cases are represented: (a) the *without caching* or base-line case where caching was disabled, (b) the *empty cache* case where caching was enabled but the cache was cleared before each repetition of the test set, and (c) the *non-empty cache* case where the cache was contained the results for all queries in the test set. The difference between the base-line and empty cache cases is due to the number of repeated queries (i.e., 26 out of 48 were repeated). The fact that the base-line case is more than twice the empty cache indicated that the overhead required for result caching is not significant. The non-empty cache case indicates the maximum potential benefit of result caching, i.e., nearly two orders of magnitude improvement in performance. Clearly this could only occur when the cache is "stacked" as in the test. However, it does help to establish an upper limit on the possible performance improvement afforded by result caching. Obviously, as the number of repeated queries increases so will the gain in performance.

Application of the IDI

Clearly, more detailed performance results need to be obtained using more exhaustive test sets. It will be particularly important to integrate the IDI with an AI system and measure its performance with a variety of different applications. We are currently using the IDI to provide a database server for the Unisys spoken language understanding system and are investigating the integration of the IDI with the *Intelligent System Server* and its Protem representation and reasoning engine,

The IDI and the ISS. Protem is a hybrid system containing both a frame-based representation system and a logic-based reasoning component. The integration of a frame-based representation system with a relational database management system is not straightforward. Our current approach labels some of the classes in the frame system as "database classes". Any knowledge base activity which searches for the instances of this class will be handed a stream of "database instances" which will be the result of a query sent to the database via the IDI. In order to avoid filling the knowledge base memory with database information, these instances are not installed as persistent knowledge base objects but exist as "light weight objects" which are garbage collected as soon as active processes stop examining them. They are also not "fully instantiated". That is, the values for the frame's roles are not necessarily installed. Instead, if an attempt is made to access their roles, additional database queries to retrieve the information will generated automatically. Once again, this information is not added as permanent knowledge base data, but only last as long as the currently active process is using it.

This approach has three advantages: it is relatively simple to implement, transparent to the user and is the key to isolating the data copy problem to cache validation as stated earlier. Once the relationship between a database class and its database tables is declared, the class and its instances can be treated as any other knowledge base objects. However, without the IDI cache implementation, it would be prohibitively slow.

The IDI ATIS Server. The second AI system that the IDI is being used to support is a spoken language interface to an *Air Travel Information System* database. In this project, spoken queries are processed by a speech recognition system and interpreted by the Unisys Pundit natural language system [Hirschman, et. al., 1989]. The resulting interpretation is translated into a IDIL query which is then sent to the *ATIS Server* for evaluation. This server is a separate process running the IDI which, in turn, submits SQL queries to an INGRES database server.

The "travel agent" domain is one in which there is a rich source of pragmatic information that can be used to infer the user's intentions underlying their queries. These intentions can be used to generate advice to the cache manager to allow it to make intelligent choices about query generalizations, pre-fetching and replacement. We currently have an initial ATIS server running and will be collecting statistics on its transactions which can then be used to define a effective advice strategy.

Conclusion

Although the implementation of the IDI is not complete, it does provide a solid foundation for easily creating a sophisticated interface to existing DBMSs. The key characteristics of IDI are efficiency, simplicity of use, and a high degree of portability which make it an ideal choice for supporting a variety of AI and related applications which require access to remote DBMSs.

Among the various extensions to the IDI that have been planned for the future, most involve the Cache Manager. At present, the implementation of the CM has been focused on efficient result caching and most other cache management functions have not been implemented. One of the first steps will be to impose a parameterized limit on the size of the cache and to implement a cache replacement strategy. Other extensions to the CM include cache validation, and the ability to perform DBMS-like operations on cache elements [O'Hare and Sheth, 1989].

If the IDI is extended so that it is capable of performing DBMS-like operations on the contents of its cache then, given an IDIL query, it will have three general courses of action which it may take to produce the results: (a) the entire IDIL query can be translated into SQL and sent to the remote DBMS for execution; (b) the entire IDIL query can be executed locally by the IDI (including simple retrieval from the cache); and (c) the IDIL query can be decomposed so that part of it is executed on the remote DBMS and part of it is executed locally by the IDI. The decision of which action to take would depend on a number of factors including the current contents of the cache and the estimated costs for each alternative.

References

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

[Bocca, 1986] 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.

[Brodie, 1988] 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.

[Ceri, et. al., 1986] 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 1986.

[Chakravarthy, et. al., 1982] 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.

[Chamberlin, et. al., 1976] D. Chamberlin, *el al.*. "SEQUEL2: A Unified Approach to Data Definition, Manipulation, and Control", *IBM Journal of R&D*, 20, 560-575, 1976.

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

[Chang and Walker, 1984] 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.

[Chimenti, et. al., 1987] 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.

[Dahl, et. al., 1990] D. Dahl, L. Norton, D. McKay, M. Linebarger and L. Hirschman, "Management and Evaluation of Interactive Dialogue in the Air Travel Information System Domain", submitted to *The DARPA Workshop on Speech and Natural Language*, June 24-27,1990, Hidden Valley, PA.

[Finin, et. al., 1989] Tim Finin, Rich Fritzson, Don Mckay, Robin McEntire, and Tony O'Hare, "The Intelligent System Server — Delivering AI to Complex Systems", *Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence – Architectures, languages and Algorithms*, March 1990.

[Fritzson and Finin, 1988] Rich Fritzson and Tim Finin, "Protem – An Integrated Expert Systems Tool", Technical Report LBS Technical Memo Number 84, Unisys Paoli Research Center, May 1988.

[Hirschman, et. al., 1989] Lynette Hirschman, Martha Palmer, John Dowding, Deborah Dahl, Marcia Linebarger, Rebecca Passonneau, François-Michel Lang, Catherine Ball, and Carl Weir. The pundit natural-language processing system. In *AI Systems in Government Conference.* Computer Society of the IEEE, March 1989.

[Intellicorp, 1987] Intellicorp, "KEEConnection: A Bridge Between Databases and Knowledge Bases", An Intellicorp Technical Article, 1987.

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

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

[Kellog, et. al., 1986] 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.

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

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

[Morris, 1988] 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.

[Motro, 1986] Amihai Motro, "Query Generalization: A Method for interpreting Null Answers", in *Expert Database Systems*, ed. L. Kerschberg, Benjamin/Cummings, Menlo Park CA, 1986.

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

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

[O'Hare, 1989] A. O'Hare, "The Intelligent Database Interface Language", Technical Report, Unisys Paoli Research Center, June 1989.

[O'Hare and Sheth, 1989] A. O'Hare and A. Sheth, "The interpreted-compiled range of AI/DB systems", *SIGMOD Record*, 18(1), March 1989.

[O'Hare and Travis, 1989] A O'Hare and L. Travis, "The KMS Inference Engine: Rationale and Design Objectives", Technical Report TM-8484/003/00, Unisys - West Coast Research Center, 1989.

[O'Hare and Sheth, 1989] Anthony B. O'Hare and Amit Sheth. *The architecture of BrAID: A system for efficient AI/DB Integration.* Technical Report PRC-LBS-8907, Unisys Paoli Research Center, June 1989.

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

[Sheth, et. al., 1988] 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, October 1988.

[Stonebreaker, et. al., 1987] 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.

[Van Buer, et. al., 1985] 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," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles CA, August 1985.