Design of EIDI: A Cache-based Interface to Integrate AI and Database Systems with Dynamism

Sheikh Sadid-Al-Hasan
Dept. of Computer Science & Engineering, BUET, Dhaka, Bangladesh, e-mail: *sadid_hasan@yahoo.com*

***Abstract*—The EIDI (Enhanced Intelligent Database Interface) integrates artificial intelligence and database systems where the AI system can access one or more databases on one or more remote database management systems (DBMSs). The interface can support a wide variety of different DBMSs with static as well as dynamic capabilities. SQL is used to communicate with remote DBMSs and the implementation of the EIDI provides a high degree of portability. The query language of the EIDI is a restricted subset of function free Horn clauses, which are translated to SQL. Results from the EIDI are returned one tuple at a time and the EIDI manages a dynamic cache of result relations to improve efficiency. The objective of this paper is to present firstly a brief overview of the area of AI/DB integration—which worked as the main motivation behind the EIDI, then a discussion on the major features of EIDI and subsequently the architecture and salient components of the EIDI.**

***Keywords*—EIDI, AI/DB integration, cache validation, automated schema, update tracker.**

I. INTRODUCTION

The EIDI (Enhanced Intelligent Database Interface) is an extension of IDI (Intelligent Database Interface) [1], an efficient access interface for artificial intelligence systems providing them with access to one or more remote database management systems (DBMS) which support SQL. The original IDI is one of the key components of the Intelligent Systems Server (ISS). The IDI has also been used for an Air Travel Information Systems that is accessed by a spoken language system implemented in Prolog [2, 4, 5]. The strong points of the IDI include the use of SQL, as it permits access to a wide variety of DBMSs with little or no modification. 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 [3]. 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. IDI manages a cache of result relations to improve efficiency. Results of queries to a DBMS are cached, improving the overall performance of the system and a query manager accesses the cache transparently. But a major shortcoming of IDI is that it does not attempt cache validation, which restricts its effectiveness when dynamic databases are considered. This deficiency limits the usage of this interface as it can only access 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. The proposed framework of EIDI will focus on this problem so that it can access dynamic DBMSs as well as those that are supported by the current implementation of IDI. In this paper, the organization and the major components of Enhanced Intelligent Database Interface (EIDI) are proposed including a brief overview of AI/DB integration and a discussion of the most significant features of EIDI.

II. OVERVIEW OF AI/DB INTEGRATION

The motivation behind EIDI largely depends on IDI whose main incentive was based on the fourth among the following approaches discussed for integrating AI and DB systems:

A. 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 [6].

B. Extending the DBMS System

This approach extends a DBMS to provide knowledge representation and reasoning capabilities. Here, the DBMS capabilities are the central concern and the AI capabilities are added in an ad hoc manner.

C. 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. 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 [7].

D. Tight Coupling

The tight coupling 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 [8, 9]. 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.

E. The Intelligent Database Interface (IDI)

The IDI is an interface that can be used to facilitate the development of AI/DB systems using the tight coupling approach. The main distinction of IDI is that, this interface to DBMSs uses cache technology and is designed to be easily integrated into various types of AI systems [10].

The EIDI is an enhanced interface that can be used to surmount the main restriction of IDI that does not provide a dynamic cache to integrate those databases that are frequently updated with the AI systems. The design of the EIDI also allows it to be used as an interface between DBMSs and other types of applications such as database browsers and general query processors.

III. THE PROPOSED ARCHITECTURE

A. Design Features

The EIDI supports several features, which simplify its use in an AI system. These include:

a) Connections to a DBMS are managed transparently so that there can be multiple active queries to the same database using a single open connection.

b) Connections to a given database are opened upon demand, i.e. at first use instead of requiring an explicit database open request.

c) Database schema information is loaded from the database either when the database is opened or when queries require schema information based upon user declarations.

d) The query interface is a logic-based language but uses user-supplied functions to declare and recognize logic variables.

e) Results of queries to a DBMS are cached, improving the overall performance system and a query manager accesses the cache transparently.

f) The update tracker periodically validates contents of the cache so that the interface can deal with frequently changed database relations.

B. Managing Connections

As discussed 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. There are two modes of interaction between an AI system and a DBMS:

a) 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.

b) 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 although it requires 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 [11].

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 although this requires 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. 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 EIDI 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.

C. Accessing Schema Information

One of the principal features of the EIDI 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 special queries to AI. The EIDI assumes the responsibility for obtaining the relevant schema information from the appropriate DBMS. This provides several significant advantages over interfaces that rely on the user to provide schema information. The only exception may occur when the schema on the DBMS is modified after the EIDI has accessed it since the EIDI caches the schema information. The EIDI provides a complex mechanism for forcing the schema information to be updated with the incorporation of the update tracker. 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.

D. The EIDI Organization

As Fig. 1 illustrates, the EIDI comprises of five major components –

a) The Schema Manager

b) The DBMS Connection Manager

c) The Update Tracker

d) The Query Manager and

e) The Cache Manager.

There are three principal types of inputs or requests to the IDI:

a) A database declaration

b) A special query written to act as an input to the AI system and subsequent retrieval requests against the result of that 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. The EIDI also supports other types of requests, e.g., access to schema information etc.

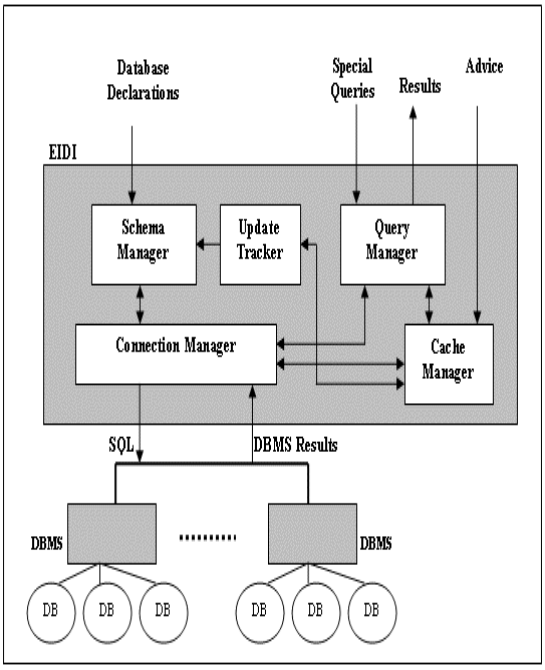

Fig. 1. The EIDI Framework

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. Time to time the SM will also deliver information to the Update Tracker (UT) in order for incorporating access to databases that are regularly updated.

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 Update Tracker (UT) is mainly responsible for dealing with dynamic databases in order to validate cache at a regular time interval. Although the Cache Manager manages part of the database that is frequently accessed, the major problem occurs when the database is changed immediately after a cache upload. The solution to remedy this problem was the main motivation of this paper and for this, a separate module, UT, is introduced. The UT makes a connection regularly with the Cache Manager to know whether the part of the result relations present currently in the cache is updated recently in its main residence in the database, if so, then the UT forces the Cache Manager to cache those again and as a part of this activity it also insists the Schema Manager to update the schema information as well if needed.

The Query Manager (QM) is responsible for processing the special queries and managing their results. Those queries are translated into SQL and 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 the special query. 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 special queries in AI 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 [1]:

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 [14].

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.

Responding - when to response the Update Tracker for updating the cache frequently? This decision tends the whole interface to cope intelligently with the dynamic databases since they are always susceptible to be changed any time.

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" [15]

Replacement - which relations should be removed when the cache becomes full?

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 proposal of EIDI successfully does cache validation with the inclusion of Update Tracker which will be an efficient solution to this problem to take account of accesses to databases that are not write-protected and updated frequently, such as several online web applications which use different AI applications accessing them etc.

IV. PERFORMANCE ISSUES

The performance of our proposed EIDI architecture is estimated considering a test set of special queries written in Common Lisp comprising 48 queries where there were 22 unique queries, i.e., each query was repeated at least once in the test set [12, 13]. 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.

Fig. 2 indicates the effects of result caching on performance. The results represent the mean processing times (in seconds) for all queries. Four different cases are represented:

a) The without caching or baseline 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.

d) The non-empty cache with dynamism where the Cache Manager co-operates with the Update Tracker for knowing the most recent updates of those parts of the database relations, which are currently resident to cache.

The difference between the baseline 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. But up to this, the third case is restricted to access only those databases that are write-protected or static.

The not-empty cache with dynamism, the fourth case evidently improves the overall usability of the interface by providing cache validation that makes it possible to work with frequently updated databases. Certainly, the process of cache validation significantly increases the performance of the overall framework while requiring a bit more time indicating the overhead associated with the co-operation between Cache Manager and the Update Tracker. 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.

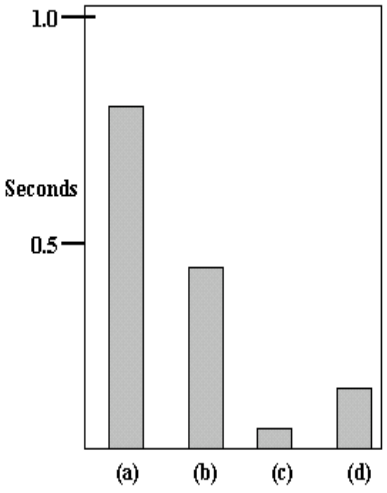

Fig. 2. Cache performance is measured for four cases

V. CONCLUSION

In this paper, we proposed the EIDI framework comprised of a new horizon to provide an effective solution to cache validation problem with the inclusion of Update Tracker, which monitors the database transactions for updates providing completeness and ease of implementation. Although the implementation of the EIDI is not complete, it does provide a solid foundation for easily creating a sophisticated interface to existing DBMSs. The key characteristics of EIDI 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. At present, the implementation of the CM has been focused on efficient result caching and most other cache management functions have not been implemented. Among the various possible extensions to the EIDI, 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 wing to the EIDI architecture may include the time coordination strategy of the Update Tracker which can be done following a sophisticated algorithm to determine the most suitable time interval to knock the Cache Manager in order for an efficient cache validation.

REFERENCES

[1] [McKay, Finin and O'Hare', et.al, 1990], Donald P. McKay and Timothy W. Finin and Anthony O'Hare', "The Intelligent Database Interface: Integrating AI and Database Systems", AAAI-90 Proceedings, 1990.

[2] [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.

[3] [Brodie, 1988J 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.

[4] [Ceri, et. al., 1986] S. Ceri, G. Gottlob, and G. Wiederhold, "Interfacing Relational Databases and Prolog Efficiently," Proc. of the lat Intl. Conf. on Expert Database Systems, South Carolina, April 1986.

[5] [Chakravarthy, et. al., 1982] U. Chakravarthy, J. Minker, and D. 'Ikan, "Interfacing Predicate Logic Languages and Relational Databases," in Proceedings of the First International Logic Programming Conference, pp. 9I-98, September 1982.

[6] [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.

[7] [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.

[8] [Chimenti, et. al., 1987] D. Chunenti, 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.

[9] [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.

[10] (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.

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

[12] [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.

[13] [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.

[14] [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.

[15] [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.