Practical Problems in Coupling Deductive Engines with Relational Databases

Abstract

Juliana Freire
Bell Labs
juliana@research.bell-labs.com

Abstract

There has been a considerable demand for applications that extend the capabilities of databases, such as data warehouses, decision support systems and knowledge discovery, to name a few. Unfortunately, adding new functionality by coupling separate components with commercial relational databases is usually a non-trivial task. One of the main reasons is the fact that the application programming interfaces of commercial relational databases do not provide adequate mechanisms that support efficient communication between client applications and the database servers. In this paper we illustrate this problem by focusing on the more specific issue of coupling deductive database query engines and relational databases.

1 Introduction

Relational database management systems (RDBMSs) are huge monolithic systems. As new technologies come along, vendors selectively add new features as an integral part of these systems, and as each vendor independently develops their version of a certain feature, applications developed for a specific RDBMS usually cannot be easily ported to a different system. Furthermore, implementing complex functionality is hard, and as a result the turnaround time between the inception of new technology and its availability in a RDBMS can be unacceptable.

Consider for instance recursive query processing. Even though significant progress has been made in area of deductive databases (DDBs) in the past 20 years, none of the available commercial systems makes full use of this technology. Besides there is a great variation on how recursive queries are supported by different RDBMSs. For example, whereas DB2 can evaluate complex forms of recursion [Cha96], Oracle only supports hierarchical queries [KL95]. At the same time, a number of research prototypes that use state of the art DDB techniques have been built, and theoretically, these could be coupled with existing RDBMSs to provide the latter with a full range of deductive capabilities. In practice however, given the available application programming interfaces (APIs), an efficient and portable interface between a DDB engine and a commercial RDBMS is very hard (and I wonder if possible) to build.

There has been considerable progress on extensible databases [CH90]. Starburst [HCL+90] and Exodus [CDF+86] are good examples of research prototypes that attempted to make it easier to customize database systems to suit user's special needs. But given that commercial RDBMSs are neither as modular as Exodus, nor make their source code available, an important factor to make these systems extensible is to make them easier to interoperate with, by providing suitable mechanisms to access their functionalities and to communicate with them.

RDBMSs provide a few different APIs through which client applications can access the database servers. It is not our intent in this paper to give the specification of an "ideal" API for RDBMSs. Rather,

Proceedings of the 5th KRDB Workshop
Seattle, WA, 31-May-1998

(A. Borgida, V. Chaudhri, M. Staudt, eds.)

http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/

we use the specific problem of coupling deductive database engines and relational databases to illustrate deficiencies of the currently available APIs that hamper the extensibility of relational systems.

The structure of the paper is as follows. We start in Section 2 with a brief introduction to DDBs. In Section 3 we describe the main requirements for an efficient interface between a DDB and a RDBMS, and report on our experience and difficulties encountered in trying to couple the a deductive query engine with a commercial RDBMS. In Section 4 we discuss desirable features that if supported by RDBMSs would lead to a simpler and more efficient DDB-RDBMS interface. We conclude in Section 5 with examples of applications other than DDBs that can benefit from improved communication mechanisms in APIs for RDBMSs.

2 Deductive Databases

Much of the success of relational databases can be attributed to the declarativeness of the query language SQL. Unfortunately, SQL is not expressive enough. There are many useful queries (e.g., queries that involve recursion) which cannot be expressed in this language. Usually, when one wants to reason about the contents of a database, it is necessary to leave the relational model by embedding SQL into a lower-level language such as C, which results in an *impedance mismatch*[1] and consequent loss of declarativeness. Deductive databases [Min88] address this problem by adopting logic programming [Llo84] or a restriction such as Datalog [Ull89] as the query language.

The problem of evaluating Datalog queries has been extensively studied. Two approaches have gained special attention: (1) magic sets [BR91], which adds goal directedness to bottom-up evaluation, and (2) tabling (or memoization) [CW96], which adds features of database evaluation to logic programming languages. These two approaches resemble each other in that they combine top-down goal orientation with bottom-up redundancy checking. In fact, under certain assumptions they have been proved to be asymptotically equivalent [Sek89]. In practice, tabling systems were shown to be an order of magnitude faster than magic-style systems for in memory queries [SSW94], and at least as efficient for queries involving external data [FSW97].

A number of research prototypes such as, LDL [CGK+90], CORAL [RSS92], Aditi [VRK+94], XSB [SSW94], are available.[2] These prototypes have been used in various applications [Ram95] and in different domains. XSB, for instance, has been used for semantic integration of information systems [PAE98], program analysis [CDS97], model checking [RRR+97], and natural language analysis [LWFS96], to name a few.

Given the availability of mature DDB systems such as XSB, and the unique advantages of DDBs in allowing non-trivial manipulation of data, DDB technology is likely to be instrumental in many emerging applications, such as middleware, decision support and knowledge discovery systems, which require support for complex queries and reasoning.

3 Interfacing a Deductive query engine with a RDBMS

There have been two main approaches to building DDB systems: integration, where a new database with deductive capabilities is built from the ground up (e.g., Aditi); and coupling, where specialized deductive engines are coupled to existing databases or storage managers (e.g., Coral and XSB).

Coupling and keeping the deductive engine and RDBMS separate has a number of advantages: deductive capabilities can be used with arbitrary RDBMSs and in a heterogeneous environment; and there are no extra overheads for applications that do not access data stored in a RDBMS. But there are also drawbacks. Unlike integrated DDBs, coupled systems incur overheads for applications that use external data, since tuples must be sent back and forth from the DDB to the database server. Thus, for coupled DDB engines to be practical for such applications, the DDB-RDBMS interface must keep communication overheads to a minimum. Besides, in order to be able to access multiple heterogeneous databases, such an interface must be portable among different databases.

In what follows we will describe the main issues to be considered when coupling a DDB query engine with a RDBMS.

Querying the Database

When external tables are accessed within a DDB, the XSB system, as do other DDBs and logic programming systems, translates Datalog fragments that access external relations into dynamic SQL queries which are sent to the database server.[3] Thus, an efficient translation that minimizes the accesses to the RDBMS is crucial to attain good performance for two main reasons: it reduces the communication costs as well as

[1] A mismatch between the data manipulation language and the host programming language.

[2] For comprehensive survey see [RU95].

[3] There are two main programming models found in the SQL standards and in RDBMSs to handle dynamic SQL queries: embedded SQL, where SQL statements are embedded in a host language such as C, and are later translated by pre-compilers into function calls to runtime libraries of the database; and call-level interfaces, where runtime library functions are directly used in the host language.

the run-time overheads for optimizing these dynamic queries. There is an array of optimization techniques that try to accomplish this goal. The following example illustrates one such technique.

Example 3.1 Consider the following program:

```
:- db_import(oracle,supplier(sid,sname),db_supplier).
:- db_import(oracle,supplies(sid,part),db_supplies).
:- db_import(oracle,order(client,part,qty),db_order).

get_supplier_id(Part,Sid) :-
    db_order(_,Part,_), db_supplies(Sid,Part).
get_supplier_name(Part,Sname) :-
    get_supplier_id(Part,Sid), db_supplier(Sid,Sname).
```

Where supplier, supplies and order are tables stored in Oracle, and each can be accessed locally in the DDB query engine through its alias (db_supplier, db_supplies, and db_order respectively). The query:

```
:- get_supplier_name(computer,X).
```

which seeks to know the names of suppliers that sell computers, will issue at least two SQL queries to the database (one for each join, in the first and second clauses). If this program is rewritten so that the superfluous temporary relation get_supplier_id is eliminated, a single query with a triple join will suffice.

```
get_supplier_name(Partid,Sname) :-
    db_order(_,Part,_),
    db_supplies(Sid,Part),
    db_supplier(Sid,Sname).
```

□

This and other high-level compile-time optimizations have been well-studied in the literature (see e.g., [CGT90]), and they do indeed improve the performance of query evaluation. However, in an implementation, another important issue that is not captured in these optimizations should be taken into account: the actual communication mechanisms — how tuples are sent back and forth between a DDB and an RDBMS. Surprisingly, this issue has been largely neglected in current systems.

DDB systems use set-at-a-time evaluation strategies which mesh well with database set-based operations such as relational joins. However, even though relational database operations are set-based, access mechanisms to relational systems are tuple-at-a-time. Consequently, there is a *communication mismatch* between the client DDB engine and the database server.

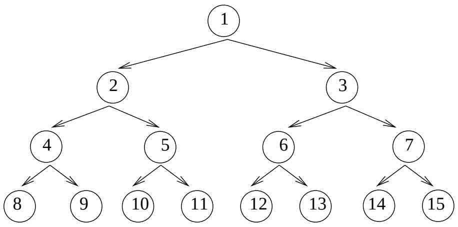
Under standard interfaces that support dynamic SQL, the results of a query are associated with a cursor. After a query is executed, the resulting tuples can then be fetched one by one from the associated cursor by the client application. For data intensive applications, sending a fetch request for each tuple is too expensive. A reasonable assumption is that database libraries should handle this transparently, by caching pre-fetched tuples at the client side to reduce the round-trips between client and server. But that is not always the case. No support for transparent caching was provided in Oracle 7, and Oracle 8 supports allows caching through its (non-standard) OCI call-level interface [Loc97] but not through embedded SQL [Mel97], which is also the case for DB2 v2 [Cha96]

Another alternative provided by some databases to reduce the communication costs is through an array interface. In Oracle, for instance, by explicitly defining arrays (one for each column to be retrieved), the client application can control to a certain extent how many tuples are fetched at a time. Just to give an idea of the performance gains from fetching multiple tuples at a time, the graph in Figure 1(a) shows the times to select 32,000 rows of a table stored in Oracle using array of varying sizes, to fetch from 1 to 64 rows at a time. Notice that there is a significant speedup. It is worth pointing out that in this experiment, client and server ran in the same machine. If data were transferred across the network the speedups could have been significantly bigger.

Joining In-Memory and External Relations

Another aspect that has been overlooked in (coupled) DDB prototypes is how to efficiently join a relation stored within the DDB (in-memory) with a table in an external database. For instance, in CORAL, even though in-memory evaluation of recursive queries is done in a set-at-a-time fashion, when joining in-memory and externally stored relations, a tuple-at-a-time nested-loop join is used [RSSS94]. In what follows, we illustrate the inefficiency of this method, and the obstacles posed by the APIs of RDBMSs to achieve a more efficient solution.

Example 3.2 Consider the following rules that define the path relation:




```
:- db_import(oracle,edge(source,target),db_edge).

:- table path/2.
path(Src,Tgt) :-
    db_edge(Src,Tgt).
path(Src,Tgt) :-
    path(Src,TgtTmp),db_edge(TgtTmp,Tgt).
```

and the query :- path(1,X) which seeks to find (recursively) all nodes reachable from node 1 in the edge table (stored in Oracle), whose contents are depicted in the graph above. Let us examine how this query is evaluated in the XSB system.[4]

XSB uses a top-down breadth-first search strategy to evaluate recursive queries that involve external data in a set-at-a-time fashion (which is equivalent to the semi-naive evaluation of a magic-transformed query [FSW97]). The query :- path(1,X) is first resolved against the non-recursive rule, which finds the nodes directly connected to node 1 (nodes 2 and 3) by selecting the desired tuples from the base table. The second rule finds nodes indirectly connected to node 1 by iteratively joining the set of tuples in the in-memory path relation derived in the previous iteration with the external edge table, until a fixpoint is reached.

The original XSB-Oracle interface compiles the second rule into a parameterized SQL select statement SELECT Tgt FROM edge WHERE Src=TgtTmp, that is issued for the values of TgtTmp from relevant tuples in the path relation. The execution of the translated query is equivalent to a nested-loop join, and it is clearly inefficient as it will result in one database access for each edge tuple (in the example, 15 select statements are issued). □

A straightforward solution would be to materialize the edge table in XSB. However, this might not be feasible for relations that do not fit in memory. And even when the table fits in-memory, if the query only uses a small subset of the tuples, materializing the whole table might lead to unnecessary communication, as it requires all tuples to be sent from the RDBMS to the DDB.

A more scalable and potentially more efficient approach is to perform actual joins instead of the multiple selects required by the original XSB-Oracle interface. At each fixpoint iteration, XSB sends the relevant set of values in the path relation to be joined with the base relation edge. In the example, this would result in 4 database accesses (instead of 15). Unfortunately, this cannot be done efficiently through the APIs of current commercial RDBMSs.

Let us go over some possible translations for the second rule of Example 3.2. A simple translation is: SELECT * FROM edge WHERE Src IN <TmpTgt>, where <TmpTgt> is the relevant set of values from the path relation. The question now is how to send the set <TmpTgt> to the database. The array interface would seem to be a natural choice, but neither DB2 nor Oracle allow arrays to be used as arguments for the IN predicate [Mel97, Cha96].[5] A more general and scalable solution is to create a temporary table in the database, populate it with the relevant tuples from the path relation, and then perform the join with the edge table.

The latter solution was implemented in the XSB system. The graph in Figure 1(b) shows that this set-at-a-time translation can lead to significantly better performance compared to the the original tuple-at-a-time translation.[6] It is worth pointing out that even though temporary tables are defined in the SQL92 standard [MS93], they are not supported in Oracle. Therefore, the overall performance of the set-at-a-time translation is hampered by the high overheads of creating and maintaining the (pseudo) temporary tables, which in some examples account for 70% of the query execution time.

4 Improving the APIs of RDBMSs

In the examples above we described some of the requirements for achieving an efficient coupling of a DDB query engine and an RDBMS: (1) the ability to communicate multiple tuples at a time back and forth from the DDB engine to the RDBMS, (2) the ability to manipulate these tuples in RDBMS in an efficient way, and (3) portability, the ability to access different RDBMSs. Below, we discuss if and how these requirements can be realized through current RDBMS APIs.

Sending/fetching tuples

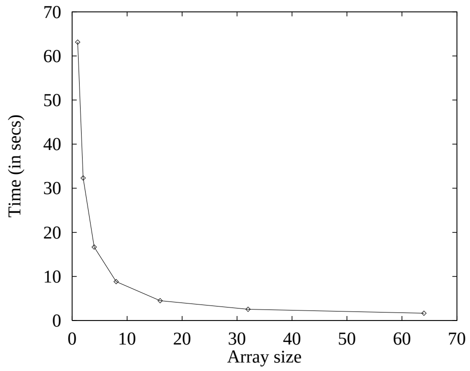
The issue sending/fetching multiple tuples at a time is partially addressed through the array interfaces provided by some RDBMSs, such as Oracle's *host arrays* [Mel97], and the *SQLExtendedFetch* facility of ODBC [Gei95]. The host arrays supported by Oracle allow client applications to explicitly control how many values are sent/fetched from the database server. We have shown in Figure 1(b) that host arrays can significantly reduce communication overheads. Nonetheless, Oracle's host arrays have a number of limitations [Mel97]:

- The values sent to the server cannot be operated on as a set. For example, an array cannot be joined with a database table.
- There is a fixed limit on the number of bytes an array can store which can not be set by the client application.

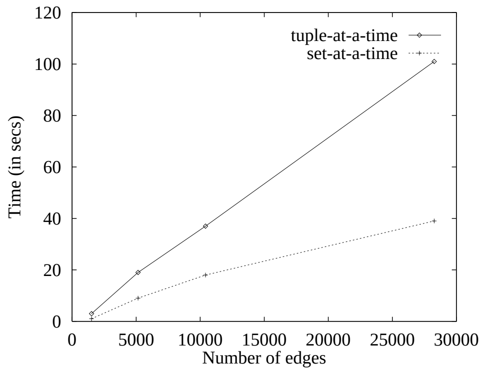
[4] In XSB, an in-memory table is created to store the tuples of relations (views) declared as tabled. By selectively "tabling" certain views, XSB can avoid redundant computation and the infinite looping of Prolog for programs such as the left-recursive transitive closure of Example 3.2.

[5] One could also generate a string with all values, but clearly this is not a scalable solution.

[6] The graphs used for this experiment were generated with Knuth's Stanford Graph Base [Knu93].



(a) *Time variation for different array sizes*



(b) *Elapsed times for different translations to evaluate the query path(words,X) using variations of the Words graph stored in Oracle*

Figure 1

- From a design point of view, one might also argue that using one array per column is quite unnatural, as most applications manipulate rows, not columns.

Even though in its latest release (version 8) Oracle provides a new array interface for its OCI API that addresses these limitations, they remain in the embedded SQL API.

In contrast to host arrays, in ODBC level 2 the client application can set how many tuples should be cached at the client side, and when a *SQLExtendedFetch* is issued, multiple tuples are automatically fetched from the server. This approach is more transparent than having to explicitly define arrays, unfortunately, it allows multiple tuples to be fetched from but not sent to the database server.

Manipulating client data efficiently

A useful feature that is missing from current RDBMS APIs is the ability to efficiently manipulate data sent from client applications. For instance, to allow a set of tuples sent from the client to be directly joined with a database table. This could also be achieved indirectly by using temporary tables together with an array interface. But as we mentioned before, even though temporary tables are defined in the SQL92 standard, they are not supported in either Oracle or DB2.

Portability

There are considerable discrepancies among different RDBMSs. Therefore, designing an efficient yet portable DDB-RDBMS interface is a major challenge. The first issue to be considered is which programming model to use. For the XSB-Oracle interface, embedded SQL was chosen for two main reasons: it is simpler to program than call-level interfaces, and easier to port among different databases.

However, there is a tradeoff between portability and efficiency. For example, in order to support efficient communication between the XSB and Oracle, the set-at-a-time XSB-Oracle interfaces uses the non-standard (thus, non-portable) host arrays provided by Oracle.

5 Discussion

We have discussed practical issues to be considered in coupling DDB engines and RDBMSs. Even though there are well-known optimizations to reduce the number of accesses to the RDBMS, the communication mechanisms between the systems have been largely neglected in coupled DDB systems. We have shown that efficient mechanisms to send sets of tuples back and forth from DDBs to the RDBMSs, as well as the ability to operate on these tuples are crucial to attain good performance. Unfortunately, adequate support for these mechanisms is not provided in the APIs of commercial relational databases such as Oracle and DB2.

The inability of RDBMSs to efficiently handle data intensive communication from/to client applications hampers their usability not only for DDB engines, but for a number of applications that require non-trivial manipulation of large amounts of data, such as workflow managers, mediators, decision support systems and data mining tools. Consider for instance a mediator wrapper that accesses multiple heterogeneous databases. If a query requires a join between relations residing in different databases, the join can be performed in one of the databases or in the mediator. In either case tuples must be transferred across the network: from one database to the other where the join is to performed, and later the results back to the mediator; or tuples of both relations are sent to the mediator. In both scenarios it is very important to be able to transfer multiple tuples at a time, and in the first scenario there is also the need (as in Example 3.2) to join the set of tuples being transferred against a table in the database.

Recent improvements in some commercial RDBMS APIs indicate that vendors are starting to recognize the need for efficient communication mechanisms between client applications and database servers. Efforts towards designing more portable interfaces are also underway [Mel95, FMMP96, Gei95], but adoption of such standards remain a rather slow process.

References

[BR91] C. Beeri and R. Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10(3):255–299, 1991.

[CDF+86] M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J.E. Richardson, and E.J. Shikita. Architecture of the EXODUS extensible DBMS. In *Proceedings of the International Workshop on Object-Oriented Database Systems*, pages 52–65, 1986.

[CDS97] M. Codish, B. Demoen, and K. Sagonas. XSB as the natural habitat for general purpose program analysis. In *Proceedings of the International Conference on Logic Programming (ICLP)*, page 416. MIT Press, 1997.

[CGK+90] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):76–90, 1990.

[CGT90] S. Ceri, G. Gotlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag, 1990.

[CH90] M. Carey and L. Haas. Extensible database management systems. *SIGMOD Record*, 19(4):54–60, 1990.

[Cha96] D. Chamberlin. *Using the New DB2*. Morgan Kaufmann, 1996.

[CW96] W. Chen and D.S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *JACM*, 43(1):20–74, January 1996.

[FMMP96] S.J. Finkelstein, N. Mattos, I. Mumick, and H Pirahesh. Expressing recursive queries in sql. Technical Report X3H2-96-075r1, ISO/IEC JTC1/SC21 WG3 DBL MCI, 1996.

[FSW97] J. Freire, T. Swift, and D.S. Warren. Taking I/O seriously: Resolution reconsidered for disk. In *Proceedings of the International Conference on Logic Programming (ICLP)*, pages 198–212, 1997.

[Gei95] K. Geiger. *Inside ODBC*. Microsoft Press, 1995.

[HCL+90] L.M. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, , and E. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, 1990.

[KL95] G. Koch and K. Loney. *Oracle The Complete Reference*. Oracle Press, 1995.

[Knu93] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. Addison Wesley, 1993.

[Llo84] J. W. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1984.

[Loc97] P. Locke. *Oracle Call Interface Programmer's Guide, Volumes 1 & 2 Release 8.0*. Oracle Corporation, 1997.

[LWFS96] R.K. Larson, D.S. Warren, J. Freire, and K. Sagonas. *Syntactica*. MIT Press, 1996.

[Mel95] J. Melton. (ISO/ANSI Working Draft) database language SQL3. Technical report, ISO - International Organization for Standardization, 1995.

[Mel97] J. Melnick. *Pro*C/C++ Precompiler Programmer's Guide Release 8.0*. Oracle Corporation, 1997.

[Min88] J. Minker. *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1988.

[MS93] J. Melton and A. Simon. *The New SQL: A Complete Guide*. Morgan Kaufmann, 1993.

[PAE98] B.J. Peterson, W.A. Andersen, and J. Engel. Knowledge Bus:Generating Application-focused Databases from Large Ontologies. In *Proceedings of the 5^{th} KRDB Workshop*, 1998.

[Ram95] R. Ramakrishnan. *Applications of Logic Databases*. Kluwer, 1995.

[RRR+97] Y. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Proceedings of Computer Aided Verification (CAV)*, pages 143–154, 1997.

[RSS92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, relations, and logic. In *Proceedings of VLDB*, pages 238–250, 1992.

[RSSS94] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The coral deductive system. *The VLDB Journal, Special Issue on Prototypes of Deductive Database Systems*, 3(2):161–210, 1994.

[RU95] R. Ramakrishnan and J. Ullman. A survey of research on deductive database systems. *Journal of Logic Programming*, 23(2):125–149, 1995.

[Sek89] H. Seki. On the power of Alexander templates. In *Proceedings of PODS*, pages 150–159, 1989.

[SSW94] K. Sagonas, T. Swift, and D.S. Warren. XSB as an efficient deductive database engine. In *Proceedings of SIGMOD*, pages 442–453, 1994.

[Ull89] J. Ullman. *Principles of Data and Knowledge-base Systems*, volume 1. Computer Science Press, 1989.

[VRK+94] J. Vaghani, K. Ramamohanarao, D.B. Kemp, Z. Somogyi, P.J. Stuckey, T.S. Leask, and J. Harland. The Aditi deductive database system. *The VLDB Journal*, 3(2):245–288, 1994.