The Aditi deductive database system

(extended abstract)

Kotagiri Ramamohanarao

rao@cs.mu.OZ.AU
Key Centre for Knowledge Based Systems
Department of Computer Science
The University of Melbourne
Parkville, 3052, Australia

Abstract

The aim of the Aditi project at the University of Melbourne is to find out what implementation methods and optimization techniques would make deductive databases competitive with current commercial relational databases. The structure of the Aditi prototype is based on a variant of the client-server model. The front end of Aditi interacts with the user exclusively in a logical language that has more expressive power than relational query languages. The back end uses relational technology for efficiency in the management of disk based data and uses some optimization algorithms especially developed for the bottom-up evaluation of logical queries involving recursion. The system has been functional for almost two years now, and has already proven its worth as a research tool. This paper outlines the structure of Aditi and presents an example in some detail.

1 Introduction

Aditi is a deductive database system that has been designed and is under continuous development at the University of Melbourne. Its purpose is to allow research into the use of logic for manipulating large amounts of data. The system provides a logic programming language interface for entering programs and queries which are evaluated by a database backend. This backend is a disk based deductive database system which supports bottom-up computation methods to evaluate queries involving recursion and function symbols. This paper presents an overview of the Aditi system and discusses current developments.

We started work on Aditi in the second quarter of 1988. A basic version of the system has been operational since July 89. We are continuously enhancing the system, adding functionality and increasing performance, but we are also using it as a research tool: several techniques employed by the current version of Aditi were first implemented and evaluated using previous versions. Other aspects of Aditi build on original research done previously by members of the Aditi team [1, 2, 6, 7, 13, 14]. (Aditi is the name of an Indian goddess; she is "the personification of the infinite" and "mother of the gods".)

Another version of this paper will appear in the Proceedings of the 1991 International Conference on Data Engineering; a much longer version is available as Technical Report 90/14 from the Department of Computer Science, University of Melbourne.

2 The structure of Aditi

Aditi is based on the client/server model found in many commercial relational database systems. Users interact with a front-end process (FE) that is regarded as a client of the system. The client communicates with a back-end process (server) that per-

DATABASE SYSTEMS FOR ADVANCED APPLICATIONS '91
Ed. A. Makinouchi
©World Scientific Publishing Co.

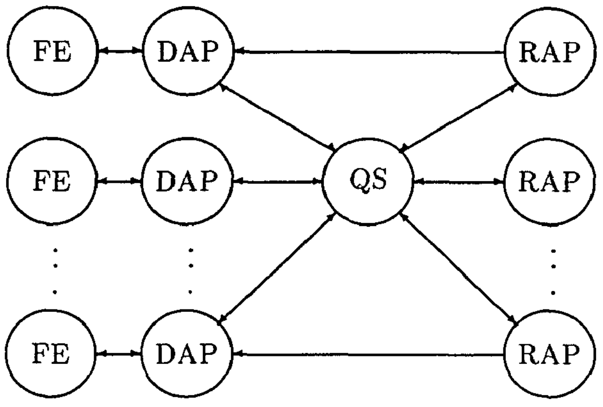

Figure 1: The structure of Aditi

forms the usual set of database operations, such as joining, merging, and subtracting relations, on behalf of the clients. Some systems have one server per client, while others have one server supporting multiple clients. Aditi is a hybrid of these two schemes: some of its server processes are dedicated to clients while others are shared by all clients.

The dedicated server process, called a Database Access Process (DAP), performs the initial authorisation clearance of the client as well as all tasks connected with query evaluation except the execution of relational algebra operations. Those operations are performed by a pool of server processes called Relational Algebra Processes (RAPs). These provide the relational operations required for query evaluation. The pool of RAPs is managed by a master process called the Query Server (QS). Figure 1 illustrates how the pieces fit together.

As a DAP evaluates a query, the relational algebra operations are sent to the QS for execution. If there is a free RAP, then the QS passes the task on to that RAP, otherwise the task is queued until a RAP becomes available. The RAP then performs the task and notifies the requesting DAP of the result. The RAP also informs the QS that it is available for another task. To reduce communication overhead, we have implemented our own IPC mechanisms using shared memory; nevertheless we intend to implement a scheme in which a RAP can be assigned exclusively to a DAP for several tasks, the RAP being returned to the process pool only when the DAP is finished with it.

Here is a short summary of the properties of the various processes.

FE The clients of Aditi are called Front End processes. When making interactive queries on the database, one would use the *query shell* as a front end. When one wants to write applications using Aditi embedded in an interpreted language such as NU-Prolog, the front end would be the (modified) NU-Prolog interpreter; When one wants to write applications using Aditi embedded in a compiled language such as C, the front end would be the application program itself.

DAP Aditi requires each Front End process to access Aditi through a Database Access Process or DAP. DAPs are responsible for database security and they oversee the execution of queries. There is one DAP per live Front End process.

QS The Query Server or QS is responsible for managing the load on the machine. In operational environments, there will be one QS per machine (in our development environment, one can set up other QSs for testing).

RAP Relational Algebra Processes or RAPs carry out relational algebra operations on behalf of the DAPs. RAPs are allocated to DAPs for the duration of one such operation. The number of RAPs that can be active at a given moment is controlled by the QS within configurable limits, so there is no necessary connection between the number of DAPs and RAPs in the system.

3 The languages of Aditi

When supplying tuples for EDB predicates, defining IDB predicates or making queries, the users interact with Aditi using only Aditi-Prolog, a variant of the logic programming language Prolog adapted for deductive databases. The DAP, however, understands only its own machine language, which is a bytecode version of RL, the Aditi relational language. The Aditi compiler, which is written in NU-Prolog, converts facts, predicate definitions and queries from Aditi-Prolog to RL; an assembler written in C converts RL to the bytecode expected by the DAP. The compiler and assembler are invoked by users when they define IDB relations and by the query shell for any except the simplest queries.

3.1 Aditi-Prolog

Aditi-Prolog is essentially just pure Prolog augmented with declarations. Some declarations tell the compiler something about the properties of the predicate: e.g. which arguments will be known when the predicate is called. Others request specific rule transformations or evaluation strategies. Among those available are naive evaluation, differential or semi-naive evaluation [2], evaluation by magic set interpreter [11], magic set transformation [3, 4], constraint propagation [6, 8], and transformations for linear rules [7, 9, 10]. Here is an example:

```
?- mode(edge(f,f)).

?- mode(path(b,f)).
?- mode(path(f,b)).
?- flag(path, 2, diff).

path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
```

The first line declares that the predicate `edge` has two arguments and that it expects to be called with both arguments free. Since this code has no definition for `edge`, it must be an EDB relation or a separately compiled IDB predicate.

The second and third lines declare that the predicate `path` has two arguments, and that it has two modes. In its first mode, it should be called with the first argument *bound* to a ground term and the second argument *free* (e.g. `?- start(X), path(X, Y)`, "what nodes Y are reachable from nodes X that appear in the `start` relation"). In the second mode, it should be called with the second argument bound and the first argument free (e.g. `?- path(X, b)`, "what nodes X is node b reachable from"). The fourth line requests differential evaluation [2] for the `path` predicate with arity 2. The last two lines are the rules defining this predicate.

3.2 Aditi relational language

RL is a simple procedural language augmented with relational algebra operations. The primitive operations of RL fall into the following classes:

- the standard relational algebra operations such as join, union, difference, select and project
- extended relational algebra operations such as union-diff, which performs a union at the same time as a difference, thus saving the overhead of scanning the input relations twice
- operations for data movement such as append (union without checking for duplicates), copy (copy the contents of a relation) and assign (copy a pointer to a relation)
- operations concerned with data structure optimization such as presorting relations
- arithmetic and relational operations on integers and floating-point numbers and the usual operations on boolean values.

The DAP sends most of the operations in the first four classes to the QS for execution by some RAP; those in the fifth class and some others (e.g. copy-pointer-to-relation) are carried out by the DAP itself.

The control structures of RL are simple: it supports only gotos, conditional branches, and procedure calls. The procedures are the key to RL. An RL program has a procedure for every mode of every predicate in the Aditi-Prolog program it was derived from. A procedure corresponding to an EDB predicate merely returns a pointer to the permanent relation (all such procedures have all their arguments free; any selections on permanent relations are done by other code). Procedures that implement IDB predicates are more complex, partly because they naturally require several steps to implement and partly because they offer more opportunities for optimization. Figure 2 is an example: it shows the RL code corresponding to the first mode of `path`.

The name of the RL procedure that implements an Aditi-Prolog predicate is derived from the name of the predicate (path), its arity (2), and its mode number (1). By convention, all these RL procedures have two arguments. The first is always a relation whose tuples represent the values of the input or bound arguments of the predicate it implements; the second is always a relation whose tuples represent the values of *all* the arguments of the predicate it implements. In this case, `init_path` has one attribute while `final_path` has two, because `path` has two arguments, only one of which is input in mode number 1. The idea is that when `path_2_1` is called, the `init_path` relation must already be known, but that `path_2_1` is responsible for determining the contents

```
procedure path_2_1(init_path, final_path)
relation init_path, final_path;
{
    relation new_path, diff_path;
    relation edge, nullary;
    bool bool1;
    int size1;

    settrel(new_path, 2);
    settrel(diff_path, 2);
    settrel(edge, 2);
    settrel(nullary, 0);

    call("edge_2_1", nullary, edge);
    copy(edge, final_path);
    copy(final_path, diff_path);
label1:
    join(edge, diff_path,
        "#(0,1)=#(1,0)", "",
        new_path, "#(0,0), #(1,1)");
    uniondiff(final_path, new_path, "",
        final_path, diff_path);
    cardinality(diff_path, size1);
    gt(size1, 0, bool1);
    test(bool1, label1);
    join(final_path, init_path,
        "#(0,0)=#(1,0)", "",
        final_path, "#(0,0), #(0,1)");
}
```

Figure 2: RL code for path

of final_path (in fact any tuples in final_path at the time that path_2_1 is called will be overwritten and thrown away).

The body of the procedure path_2_1 begins with the declaration of some local variables, and continues with the creation of (empty) relations of various arities to serve as the initial values of the relation-valued variables (settrel stands for set temporary relation).

The next two lines implement the first rule of the path predicate. The first line calls the RL procedure for the predicate edge (arity 2, mode number 1) with a zero-arity input relation and puts the result in edge. Since edge is an EDB predicate, its data is stored in a permanent relation, and edge will now contain a pointer to this relation. The second line copies the contents of edge into the relation final_path (like C, RL permits the use of parameters as temporaries, even though RL passes parameters by reference).

The rest of the procedure except for the final line implements the second rule of path. The loop invariant is that final_path holds the path facts currently known to be true, new_path holds the path facts discovered in the current iteration, and diff_path holds those facts from new_path that were not discovered in previous iterations. final_path was initialized by the first rule, diff_path is initialized in the line before the loop, while new_path is computed during the loop, The loop body starts out by implementing the join implicit in the conjunction edge(X, Z), path(Z, Y) by joining edge and diff_path with the join condition that the second argument of the former be equal to the first argument of the latter. (As both input relations and arguments are numbered from zero, the notation #(0,1) refers to input relation number 0 (i.e. edge) and argument number 1 (i.e. Z).) The join condition is split into two parts; the first one contains conditions that are useful for indexing while the second contains conditions that are not (in this case the second part is empty). It is the responsibility of the compiler to ensure that the first part is appropriate for whatever kind of indexing is available for the relations to which it refers.

The result of the join is a relation with three attributes, representing the variables X, Y and Z. Since the head of the second rule contains only X and Y, the result is projected onto X and Y (#(0,0) and #(1,1) respectively) before it is assigned to relation new_path, which thus contains tuples corresponding to the path facts we have discovered in this iteration. We then put any tuples in new_path that were not in final_path into the relation diff_path, and add all tuples in new_path to the relation final_path, maintaining the loop invariant. The uniondiff instruction carries out both these operations at the same time (its third argument allows one to specify arguments on which the output relations should be sorted; this capability is not used in this example).

At the end of the loop, we put the cardinality of the diff_path relation into the integer variable size1, and test whether this number is greater than zero. If it is, we go back to the start of the loop at label1.

When the loop exits, final_path contains the entire path relation. The final join of final_path with init_path, deletes from final_path all tuples whose first arguments are not in init_path, leaving just those the user asked for. The wastefulness of this should be self-evident. Fortunately, there are optimization methods that compute only the required subset of the path relation, thus avoiding this inefficiency [16]. Aditi implements many of these optimizations; we chose to show the unoptimized version for the sake of exposition.

3.3 The compiler

The compiler that turns programs written in Aditi-Prolog into RL is written in NU-Prolog [15]. Unlike most compilers, it represents programs in not one but two intermediate languages, which we call HDS and LDS (for "high-level data structure" and "low-level data structure" respectively). HDS provides an easy-to-manipulate representation of Prolog rules while LDS provides an easy-to-manipulate representation of RL programs. The compiler has three main stages: Aditi-Prolog to HDS, HDS to LDS, and LDS to RL, in addition to optional optimization stages that transform HDS to HDS or LDS to LDS.

The first main stage, Aditi-Prolog to HDS, is concerned mainly with parsing the input and filling in the slots in the HDS representation of the program (most of these slots contain information for HDS to HDS optimizations). Some of the tasks required for the latter are trivial, such as finding the scopes of all variables, while others can have great impact on the performance of the final code, such as selecting an appropriate sideways information-passing strategy or SIP [17].

The HDS to HDS level is where the compiler implements the optimizations that are defined in terms of source-to-source transformations. These optimizations include magic set transformation, counting set transformation [3, 4], constraint propagation [6, 8], and context transformations for linear rules [7].

The second main stage, HDS to LDS, is responsible for converting a predicate calculus oriented representation of the program into a representation geared to relational algebra operations. Among other things, this requires the transformation of recursive Prolog rules into RL procedures containing iteration. If several predicates are mutually recursive, then the compiler generates a single procedure containing one big iteration that computes values for all these predicates. It also generates an interface procedure for each predicate involved; these call the procedure containing the iteration and select the data they need from it.

The translation from HDS to LDS can take any one of several different paths. The five paths currently implemented are naive and differential implementations of the standard bottom-up interpreter, naive and differential implementations of the so-called magic set interpreter [11], and a differential implementation of a bottom-up interpreter with predicate rule ordering [12]. Each of these implementation schemes is good in some circumstances and bad in others; it is an open research problem to find the domains of optimality of each. This is a problem we intend to pursue with the help of Aditi.

LDS to LDS optimizations come from the programming language and relational database literature (in contrast to HDS to HDS optimizations, which come mainly from the logic programming/deductive database literature). They include such techniques as common subexpression elimination, loop invariant removal, moving selections before joins, and the intelligent exploitation of any available indexing.

The third main stage, LDS to RL, is necessary because the aim of LDS is easy optimization while the aim of RL is easy assembly and fast execution. First, LDS has if-then-else and while loops, whereas RL has labels and gotos; this stage converts the former into the latter. Second, LDS has no notion of the bundling of several operations together as occurs with pre-select and post-project as well as with operations such as union-diff; this stage can convert a sequence of LDS operations into a single RL instruction using peephole optimization techniques.

The input of the compiler at present must contain declarations specifying the set of optimizations to be applied to each predicate. Although we intend to add a module to our compiler that would try to determine the best set of optimizations automatically, the current setup gives us the control we now need in our experiments concerning the performance of various optimization techniques. The data to be produced by these experiments will in fact be vital in the implementation of an automatic optimization module.

The output of the compiler is human readable RL code whose syntax and structure resemble that of C programs. The translation of RL into the bytecode needed by the DAP is the task of a small, fast assembler written in C. This design makes it convenient to inspect the output of the compiler, making debugging the compiler easier. It also allows us to write RL code ourselves. This allows us to exercise much of Aditi even without a working compiler, which was very important when the compiler itself was under development and hence sometimes not working correctly. It also allows us to try out hand optimized queries on the system; we can thus experiment with optimizations before deciding to incorporate them in the compiler.

4 Further work

A prototype of Aditi is functional and we are able to evaluate most queries, but much research and development remains to be done. We outline briefly some of the more important areas we are currently examining.

Transactions: At present, Aditi has no transaction mechanism, although some of the hooks required are present. A single transaction may require the cooperation of several processes (the client DAP and any RAPs used); we need to research concurrency control methods that allow this. We are also looking at how best to resolve simultaneous updates. As recursive computations can take long periods of time, overlapping updates are more likely in deductive databases than in relational databases. We do not want to lock relations for long periods because this can drastically reduce concurrency. However, optimistic methods that force restarts of computations could cause starvation and waste too many computation resources. We would prefer a hybrid solution that minimises these problems.

Parallelism: Even the earliest versions of Aditi could execute different queries from different users in parallel. We are now working on the exploitation of parallelism *within* the evaluation of each query. Coarse grained parallelism can be exploited at the RL level. The non-recursive rules of a predicate can be evaluated independently, and the recursive rules belonging to a set of mutually recursive predicates can be evaluated independently at each iteration; this is OR-parallelism. Similarly, some rules contain operations that do not use each other's outputs and therefore can be evaluated independently; this is AND-parallelism. To take advantage of these opportunities, we have extended RL to allow RAP operations to execute in parallel, and modified the standard DAP to support these extensions by maintaining a context for each thread of parallel execution and coordinating several RAPs. The compiler now emits code that exploits OR-parallelism; we need to extend it to exploit AND-parallelism, and we need to further evaluate and tune our implementation. As far as fine grained parallelism is concerned, we are currently implementing a parallel hash-join algorithm, and we also plan to implement parallelized versions of the other relational algebra operations in the future.

Aggregates and negation: The compiler can currently generate code for programs which are stratified through negation. Soon this will be extended to include programs which are also stratified through aggregation operations such as *count, sum, max* and *min.* We are investigating what modifications to the compiler would be required for dealing with non-stratified programs for both negation and aggregation. In particular, we conjecture that a compiler can generate code for sufficiently stratified programs [5] with little or no alterations required of the interpreter or the relational back-end.

Mixed tuple-at-a-time and set-at-a-time: An advantage of a deductive database system over a logic programming system, is its ability to use relational database techniques for performing computations involving joins with large relations. However, there are often situations where it is far more efficient to perform some computations in a tuple at a time manner. For example, predicates for list manipulation such as list reverse and append should be compiled into code which takes each tuple in the input relation, performs the required list manipulation on that tuple, and then places it in the output relation. This avoids the I/O involved in maintaining the intermediate relations containing the sublists generated by the bottom-up computation of such predicates. Predicates to be computed tuple at a time can be hidden from set at a time computations by encapsulating them in RL procedures that call NU-Prolog to perform their evaluation.

Rule Transformations: We are continuing our investigations into general rule transformation techniques, building on our experience with techniques such as magic set interpreters [11], constraint propagation [6] and special optimizations for linear recursions [7]. We aim to include a strategy selection module in the compiler to choose intelligently the set of rule transformations to be applied to each rule.

Raw input-output: Two of the major sources of overhead in the present system are the double copying of data (disk to Unix buffer cache to Aditi and vice versa) and the Unix system calls required for opening and closing files containing relations and indexes. We therefore plan to move to an implementation based on a raw filesystem.

5 Summary

We have presented an overview of the structure of Aditi, a disk-based deductive database system under continuous development at the University of Melbourne.

Users interact with Aditi using a variant of Prolog, a logic programming language that makes it easy to write applications involving recursion and function symbols. Aditi's internal operations are based on relational technology, but the system also employs several optimizations specific to deductive databases. Examples include differential evaluation, magic set transformation, magic set interpreter, constraint propagation, and context transformation for linear rules. Several of these were developed at the University of Melbourne [1, 2, 6, 7, 11].

We are currently using Aditi as a tool for research into deductive databases; it has been the vehicle for the development and evaluation of several of the above optimization techniques. We aim to eventually use Aditi to prove that deductive database systems can achieve performance comparable to that of commercial relational database systems.

Acknowledgements

Jayen Vaghani was the main implementor of Aditi: he wrote the query server and the most versions of the DAP and RAP. David Kemp and Kim Marriott wrote a prototype compiler, which has now been replaced by the one described in this paper. Peter Stuckey wrote the new, extensible compiler based on a design by Zoltan Somogyi. John Shepherd wrote the RL assembler and some of the indexing methods; David Keegel wrote the query shell; Tim Leask wrote a fast message passing package and parallelized the DAP; Warwick Harvey added support for function symbols; and Jeff Schultz served as our NU-Prolog guru.

This research was supported by a grant from the Australian Research Council through the Machine Intelligence Project.

References

[1] I. Balbin, G. S. Port, and K. Ramamohanarao. Magic set computations for stratified databases. *Journal of Logic Programming*, 1990. To appear.

[2] I. Balbin and K. Ramamohanarao. A generalization of the differential approach to recursive query evaluation. *Journal of Logic Programming*, 4(3):259–262, September 1987.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the Fifth Symposium on Principles of Database Systems*, pages 1–15, Washington DC., 1986.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the Sixth ACM Symposium on Principles of Database Systems*, pages 269–283, San Diego, California, March 1987.

[5] P. M. Dung and K. Kanchanasut. A fixpoint approach to declarative semantics of logic programs. In *Proceedings of the First North American Conference on Logic Programming*, pages 604–625, Cleveland, Ohio, October 1989.

[6] D. B. Kemp, K. Ramamohanarao, I. Balbin, and K. Meenakshi. Propagating constraints in recursive deductive databases. In *Proceedings of the First North American Conference on Logic Programming*, pages 981–998, Cleveland, Ohio, October 1989.

[7] D. B. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left-, and multi-linear rule transformations that maintain context information. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 380–391, Brisbane, Australia, August 1990.

[8] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *Proceedings of the Ninth Symposium on Principles of Database Systems*, pages 314–330, Nashville, Tennessee, April 1990.

[9] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of ACM SIGMOD '89*, pages 235–242, 1989.

[10] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Argument reduction by factoring. In *Proceedings of the Fifteenth Conference on Very Large Data Bases*, pages 173–182, Amsterdam, The Netherlands, August 1989.

[11] G. Port, I. Balbin, K. Meenakshi, and K. Ramamohanarao. Magic sets made simple. Technical Report 88/20, Department of Computer Science, University of Melbourne, Melbourne, Australia, Submitted for publication.

[12] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Rule ordering in bottom-up fixpoint evaluation of logic programs. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 359–371, Brisbane, Australia, August 1990.

[13] K. Ramamohanarao and J. Shepherd. A superimposed codeword indexing scheme for very large Prolog databases. In *Proceedings of the Third International Conference on Logic Programming*, pages 569–576, London, England, July 1986.

[14] K. Ramamohanarao, J. Shepherd, I. Balbin, G. Port, L. Naish, J. Thom, J. Zobel, and P. Dart. The NU-Prolog deductive database system. In P. Gray and R. Lucas, editors, *Prolog and databases*, pages 212–250. Ellis Horwood, Chicester, England, 1988.

[15] J. A. Thom and J. A. Zobel. NU-Prolog reference manual, version 1.3. Technical report, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1988.

[16] J. D. Ullman. *Principles of knowledge-base systems, volume 2.* Computer Science Press, Rockville, Maryland, 1989.

[17] J. D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, September 1985.