VLDB Journal, 3, 245-288 (1994). Invited contribution. Hans-J. Schek, Editor
©VLDB

The Aditi Deductive Database System

Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, James Harland

Received April, 1993; revised version December, 1993

Abstract. Deductive databases generalize relational databases by providing support for recursive views and non-atomic data. Aditi is a deductive system based on the client-server model; it is inherently multi-user and capable of exploiting parallelism on shared-memory multiprocessors. The back-end uses relational technology for efficiency in the management of disk-based data and uses optimization algorithms especially developed for the bottom-up evaluation of logical queries involving recursion. The front-end interacts with the user in a logical language that has more expressive power than relational query languages. We present the structure of Aditi, discuss its components in some detail, and present performance figures.

Key Words. Logic, implementation, multi-user, parallelism, relational database.

1. Introduction

Deductive databases are logic programming systems designed for applications with large quantities of data, the kinds of applications that are at present programmed in SQL embedded in a host language such as C. Deductive databases generalize relational databases by exploiting the expressive power of (potentially recursive) logical rules and of non-atomic data structures, greatly simplifying the task of application programmers.

The ultimate goal of the Aditi project at the University of Melbourne is to show that this generalization does not have to compromise performance. Applications that can run on relational systems should run on deductive systems with similar performance while being substantially easier to develop, debug, and maintain. We

Jayen Vaghani, BSc (hons), is Systems Programmer; Kotagiri Ramamohanarao, Ph.D., is Professor; David B. Kemp, Ph.D., is Postdoctoral Research Fellow; Zoltan Somogyi, Ph.D., is Lecturer; Peter J. Stuckey, Ph.D., is Senior Lecturer; Tim S. Leask, BSc (hons), is Systems Programmer; and James Harland, Ph.D., is Lecturer; Collaborative Information Technology Research Institute, Department of Computer Science, University of Melbourne and RMIT, Parkville, 3052 Victoria, Australia.

keep performance for traditional applications competitive by using conventional relational technology whenever possible; we keep programming costs down by allowing developers to work at a level much higher than SQL and other traditional query languages. At this level, many problems can be solved without resorting to a host language. The higher level also allows the system to automatically perform better global optimization than is currently possible, because the users who currently do global optimization cannot keep in mind all the relevant details at the same time. This is particularly important for the most sophisticated applications, such as expert systems that work on large amounts of data. Therefore, deductive databases can make feasible complex applications that were not feasible before.

The project is named after Aditi, the goddess who is "the personification of the infinite" and "mother of the gods" in Indian mythology. The system has five main characteristics which together distinguish it from other deductive databases. First, it is disk-based to allow relations to exceed the size of main memory. Second, it supports concurrent access by multiple users. Third, it exploits parallelism at several levels. Fourth, it allows the storage of terms containing function symbols. Fifth, although it is oriented towards bottom-up computation like most other deductive databases, it also supports top-down evaluation of predicates.

We started work on Aditi in the second quarter of 1988, and had a basic version operational since July 1989. From then on we have continuously enhanced the system, adding functionality and increasing performance. We also use Aditi as a research tool, a platform on which to implement and evaluate new query evaluation algorithms and optimization techniques (Kemp et al., 1989, 1990, 1991, 1992; Balbin et al., 1991; Harland and Ramamohanarao, 1993; Kemp and Stuckey, 1993).

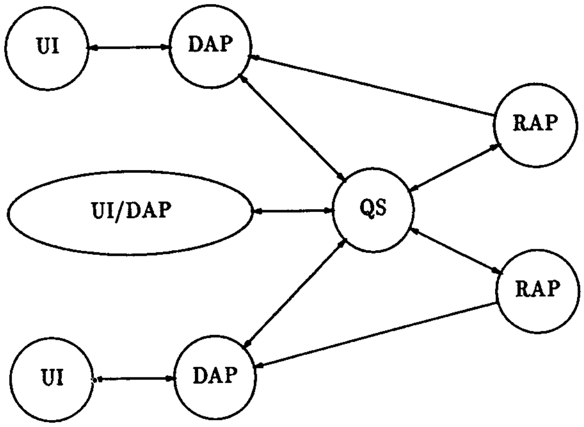
Interested researchers can obtain a beta-test copy of Aditi under a no-cost license. The distribution includes two text-based interfaces that accept Aditi-Prolog and SQL respectively, a graphical user interface, and a programming interface to NU-Prolog. The distribution is in binary form for Sun SPARC-based machines running SunOS 4.1.2 or higher and Silicon Graphics Power series multiprocessor machines running IRIX 4.0 or higher.

The structure of this article is as follows. The rest of this section introduces the main components of Aditi. Section 2 shows the structure of its databases and relations. Section 3 describes the usual Aditi clients: textual, graphical, and programming interfaces, and their languages. Section 4 details the server processes and the low-level modules they use. Section 5 gives performance figures. Section 6 compares Aditi with other systems. Section 7 concludes with some future directions.

1.1 The Structure of Aditi

Aditi is based on the client/server model found in many commercial relational database systems. In this model, users interact with a user-interface process, and this client communicates with a back-end server process that performs the usual 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

Figure 1. The structure of Aditi



server supporting multiple clients. In Aditi, each client has one server process dedicated to it, but it also shares some server processes with other clients. Figure 1 illustrates how the pieces fit together. The responsibilities of the three types of server processes are as follows:

DAP	Each Database Access Process (DAP) is dedicated to a client process. DAPs are responsible for authenticating their clients to the rest of Aditi, and for controlling the evaluation of the queries of their clients.
RAP	Relational Algebra Processes (RAPs) carry out relational algebra operations on behalf of the DAPs. RAPs are allocated to DAPs for the duration of one such operation.
QS	The Query Server (QS) is responsible for managing the load placed by Aditi on the host machine by controlling the number of RAPs active at any time. In operational environments, there should be one QS per machine. However, in a development environment one can set up multiple QSs; each QS supervises an entirely separate instance of Aditi.

When a client wants service from Aditi, it sends a command to its DAP. The DAP executes some types of commands by itself and passes the other types (mostly relational algebra operations) to the QS. The QS queues the task until a RAP becomes available and then passes the task on to a free RAP for execution. At completion, the RAP sends the result to the requesting DAP and informs the QS that it is available for another task.

2. Databases and Relations

Like other database management systems (DBMSs), Aditi requires every relation to belong to a database, but it also allows a machine to accommodate several databases.

We expect a typical arrangement to be a main production database containing detailed information about some enterprise mission, and several smaller databases in which individual users or groups record summary information, hypothetical data, test data and other information of interest only to them. The design of Aditi allows one query to gather together information from several databases, and allows users to put the result in yet another database if they wish.

2.1 Base Relations

Aditi base relations have a fixed arity, but attributes currently do not have associated types; any ground term can appear in any attribute position of any relation. Each relation has a sequence of one or more attributes as its key. If requested, Aditi will ensure that a relation has at most one tuple with a given combination of values for the key attributes.

The creator of a base relation can currently choose between four storage structures. The simplest storage structure does no indexing. The data are stored as an unsorted sequence of tuples. Although obviously not suited for large relations, this storage structure has very low overheads and is quite appropriate for small, static or mostly-static relations.

The second storage structure has B-tree indexing. The data are stored as a sequence of tuples, and a B-tree sorted on the key attributes of the relation serves as an index into this sequence; the sequence can be sorted if the user requests. The B-tree index is of a variable-bucket type. The relation's creator specifies two numbers N and B, and the node size can then vary from NB bytes to (2N-1)B bytes. The two objectives of this somewhat unconventional design are high space utilization and low overhead for relations that are created once and then used more or less unchanged several times, a usage pattern characteristic of temporary relations in deductive databases and shared by some permanent relations as well.

Tuples in the data file are initially all contiguous; holes appear only when a tuple is deleted. Aditi keeps track of these holes in a freelist, and newly inserted tuples are put into holes whenever possible. The data file of temporary relations will thus always have 100% space utilization. As for the index, the variable bucket size adjusts the amount of storage allocated to a node to the amount of storage required by the node, resulting in space utilization factors in the 90-95% range. The space efficiency of our B-tree structure reduces I/O costs for some usage patterns important to deductive databases (full scans of small-to-medium sized relations). For access patterns where clustering is important, a conventional B-tree or a multikey hashed file (see below) would be more effective.

During the creation of a relation, our B-tree structure reduces overhead by never requiring tuples in the data file to be moved; the data stay where they are even when the index node pointing to them is split. There is an overhead associated with the management of the bucket size, but our tests have shown it to be minimal. In any case, the relation creator can avoid this overhead (at the expense of worsening space utilization) by setting N equal to 1 and thus requesting fixed-size buckets.

The third storage structure has a superimposed codeword index. The data are stored as a sequence of tuples where each tuple fits entirely into one *segment*. The index has a bit string for each segment. This bit string is the segment descriptor, and it is calculated by ORing together the tuple descriptors of all the tuples in the segment. Tuple descriptors in turn are derived by hashing the values of the attributes of the tuple and using each hash value to select a fixed number of bits to be set.

When the same technique is used to generate a descriptor for a *partial-match* query on the relation (i.e., one that specifies values for some attributes but not others), it is only necessary to set the bits corresponding to the values of the specified attributes. Therefore the query descriptor will have fewer 1s than tuple descriptors. The key observations are that a tuple cannot match the query unless the tuple's descriptor has 1s in all positions where the query descriptor has 1s, and that a segment cannot contain a matching tuple unless the segment's descriptor has 1s in all positions where the query descriptor has 1s.

The first step in the retrieval process is therefore to apply this test to all the segment descriptors. Because the test looks at only the bits that are set in the query descriptor, and does not look at all the bits of segment descriptors, the index stores the segment descriptors in a bit-slice fashion, with bit 0 of all descriptors first, bit 1 of all descriptors next, and so on. This clusters together all the descriptor bits needed in the test and reduces both I/O and computation costs. However, the contiguous placement of the bit slices also places an upper bound on the number of segments of the relation. Since segments have no overflow pages in this storage method, this implies a static maximum size for the relation. This is why we call this storage structure the static superimposed codeword index (ssimc).

The ssimc removes the need to retrieve pages that do not contain tuples matching the query, but these tuples themselves can be scattered throughout the data file. To improve clustering, and thus I/O performance and memory consumption, whenever Aditi inserts a tuple into an ssimc relation, it interleaves some bits from the hash values of the tuple's attributes (which it must compute anyway) to form a segment number, and inserts the tuple into the indicated segment whenever possible. If that segment is full, Aditi tries to find another segment with sufficient space. If none exists, the insertion fails.

The fourth storage structure, the dynamic superimposed codeword index (dsimc), is specifically designed for dynamic relations, in fact for relations whose size may fluctuate by several orders of magnitude. It is very similar to ssimc, but there are two key differences. First, dsimc stores all the bits of a descriptor together, removing the limit on the size of relations. Second, whereas ssimc relies mainly on superimposed codewords and uses multikey hashing as a tuple-placement heuristic, dsimc relies mainly on multikey linear hashing and uses superimposed codewords as a filtering heuristic. In fact, one can think of dsimc as multikey hashing; before retrieving a data page, the system checks the page's descriptor to see whether the page contains any potentially matching tuples. Much of the time the answer will be "no," and in such cases we can avoid reading that data page and any associated overflow pages. The cost we pay, reading the descriptor, is small because the descriptor is small, and because it is clustered with other descriptors that the same query is likely to need.

Both the ssimc and dsimc structures require relation creators to specify some information from which they compute the parameters of the superimposed coding and multikey hashing algorithms. For a fuller explanation of research on techniques for deriving the optimal values of these parameters, see Ramamohanarao and Shepherd (1986 for ssimc; 1990 for dsimc); these papers have the full details of the two storage structures as well. As we will show in Section 4.4, Aditi's design allows other storage structures to be added later.

The creation of a relation begins with the creation of a schema (unless an appropriate schema already exists). The following command creates a schema named `flightdist_schema` of arity 3, the first two arguments forming a key, and with the flexible tuple structure (Section 4.5).

```
1% newschema flightdist_schema 3 flex 1,2
```

This command then creates the relation `flightdist` with this schema, using B-tree indexing with fixed 1 kilobyte nodes and with no duplicates allowed:

```
2% newrel flightdist flightdist_schema btree 1 1024 0
```

One can then insert facts into this relation via the command

```
3% newtups flightdist 3
sydney, honolulu, 8165
honolulu, toronto, 7464
...
```

2.2 Derived Relations

So far we have talked only about base relations. However, the power of deductive databases lies in derived relations, relations that can infer information at run-time. In Aditi, users define derived relations (also called derived *predicates*) by writing programs in Aditi-Prolog, a pure (declarative) variant of Prolog augmented with declarations. Some declarations tell the compiler 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. The file `stops.al` is an example:

```
?- mode(stops(f,f,f)).
?- flag(stops, 3, diff).

stops(Origin, Destination, []) :-
```

```
    flight(Origin, Destination).
stops(Origin, Destination, [Stop|Stoplist]) :-
    flight(Origin, Stop),
    stops(Stop, Destination, Stoplist),
    not in_list(Stop, Stoplist).

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

in_list(Head, [Head|_Tail]).
in_list(Item, [Head|Tail]) :-
    in_list(Item, Tail).
```

This code fragment defines two predicates (derived relations): `stops` gives all possible routes between all city-pairs, and represents routes as lists of intermediate stops; `in_list` checks whether its first argument is a member of the list that is its second argument.

The lines beginning with `?-` are declarations to the compiler (declarations in Prolog-like languages traditionally look like queries because they also cause immediate action by the compiler). The first mode declaration states that `stops` supports queries such as `?- stops(From, To, Stoplist)` in which all arguments are *free* variables. The second mode declaration states that `in_list` supports queries such as `?- in_list(City, [sydney, honolulu])` in which the first argument is a free variable but the second argument is *bound* to a completely specified (i.e., *ground*) term.

In general, every relation must have one or more mode declarations; base relations always have one mode consisting of all f's. A query matches a mode declaration if it has ground terms in all positions that the mode declares to be b (bound); the query may have any term in positions the mode declares to be f (free). Aditi allows a query on a relation only if it matches one of the modes of that relation. For example, since there is no "f,f" mode for `in_list`, Aditi will reject the query `?- in_list(City, Stoplist)`.

There are two reasons for requiring mode declarations:

1. Some predicates have an infinite number of solutions unless certain arguments are specified in the call.
2. Most optimization methods require that the optimizer know which arguments of each call are known when the call is processed.

It is the flag declarations that tell the compiler what optimizations and evaluation algorithms to use. With the flags shown above, Aditi will use differential (semi-naive) evaluation when answering queries on both relations; for `in_list`, it will use the magic set optimization as well. (Actually, the diff flags can be omitted, as differential evaluation is the default.)

The set of flags supported by the current version of Aditi allows users to request

- naive evaluation
- differential or semi-naive evaluation (Balbin and Ramamohanarao, 1987)
- predicate semi-naive evaluation (Ramakrishnan et al., 1990)
- evaluation by magic set interpreter (Port et al., 1990)
- the magic set transformation (Bancilhon et al., 1986; Beeri and Ramakrishnan, 1987)
- the supplementary magic set transformation (Sacca and Zaniolo, 1987)
- the context transformations for linear rules (Kemp et al., 1990)
- parallel evaluation (Leask et al., 1991)
- top-down tuple-at-a-time evaluation (Section 4.1)

This is quite a long list. What's more, users may specify several flags for one predicate. For example, one can request differential evaluation of a magic set transformed program, as `in_list` above does, or one can ask for a context-transformed program to be evaluated using a parallel implementation of the predicate semi-naive algorithm. However, not all combinations of flags make sense (e.g., the various magic set variants and the context transformation are all mutually exclusive).

The flexibility offered by the flag mechanism in selecting the set of optimizations to be applied to a predicate is essential to us as researchers when we investigate optimization methods and evaluation algorithms. Eventually, of course, we intend to distill the results of these investigations into code that *automatically* selects the best set of optimizations for each predicate in the program (as in the strategy module of Morris et al., 1987).

Aditi-Prolog programs may include disjunction and negation. Like most deductive databases, Aditi currently supports only stratified forms of negation. In the future this may change, since we have developed a practical algorithm for computing answers to queries even in the presence of unstratified negation (Kemp et al., 1991, 1992). This algorithm works on a magic-transformed program where each predicate has two slightly different versions. At each iteration, the algorithm uses one set of versions to compute a set of definitely true facts, the other set to compute a set of possibly true facts. The next iteration can then use the complement of the set of possibly true facts as the set of definitely false facts it needs to perform negation.

Aditi-Prolog supports the usual set of aggregate operations. Given an atomic query, one can ask for the minimum or maximum value of some attribute in the solution, the sum of the values of some attributes, the number of solutions, or a list of those solutions. In addition, one can ask for the solution to the query to be divided into groups based on the values of some attributes before the aggregate operation is carried out. For instance, the query

```
?- aggregate(Max = max(Dist), [], flightdist(_From, _To, Dist)),
   flightdist(From, To, Max).
```

will find the length of the longest flight in the database and print the origin, destination, and length of the few flights (probably one) that have this distance. On the other hand, the query

```
?- aggregate(Max = max(Dist), [From], flightdist(From, _To, Dist)),
     flightdist(From, To, Max).
```

will print the details of at least one flight from each city, because the groupby list tells Aditi to divide the solutions of the flightdist relation into groups, one group per starting point, and then find the longest flight(s) in each group.

When aggregation is part of an Aditi-Prolog program, it must respect stratification as well as negation. For further details of the language, see the Aditi-Prolog language manual (Harland et al., 1992*a*).

The Aditi-Prolog compiler (apc) interface is intentionally similar to the interface of other compilers on Unix systems (e.g., cc). As arguments, one just names the files to be compiled:

```
4% apc stops.al
```

The resulting object file, `stops.ro` can be tested by loading it manually into the query shell (Section 3.1). When the programmer is satisfied that the predicates in the file work correctly, he or she can make them a part of the database by executing the command

```
5% newderived stops.ro
```

3. The Clients of Aditi

There are three kinds of user interfaces to Aditi.

1. Textual user interfaces that interact with the user by reading in queries and printing out their answers. Sections 3.1 and 3.2 describe two such programs, which we call *query shells*, that take queries expressed in Aditi-Prolog and SQL respectively.
2. Graphical user interfaces that allow users to pick operations from menus. Section 3.3 introduces the main features of xqsh, Aditi's graphical interface.
3. Application programs written in a host language that access Aditi through an embedded programming interface. Section 3.4 outlines one such interface, the one that allows NU-Prolog programs to call Aditi.

All three kinds of interfaces actually send to the DAP queries expressed in a very low level language called RL. Section 3.5 describes this language, while Section 3.6 describes the compiler that converts programs and queries from Aditi-Prolog to RL.

3.1 The Aditi-Prolog Query Shell

The Aditi-Prolog query shell should be instantly familiar to anybody who has used any Prolog interpreter, because it follows the conventions established by traditional terminal interfaces to Prolog interpreters. Its main loop prints a prompt, waits for the user to type in an Aditi-Prolog query, and evaluates the query. A simple example:

```
1 <- hq(Airline, Headquarters).

Answer Set for Airline, Headquarters:

(1)  cathay_pacific, city(hongkong)
(2)  qantas, city(sydney, australia)
(3)  lufthansa, airport(frankfurt)
```

The main difference from traditional Prolog interpreters is that the query shell prints all answers to the query at once. The query shell accepts any Aditi-Prolog query, including those with disjunction, negation, aggregation, or any combination of these. To answer a query, the query shell transforms the query into a rule whose body is the query itself and whose head is an atom with a new predicate symbol whose arguments are the variables the user is interested in: these are either the variables before the colon (if there is one) or all the variables in the query. The query shell then uses the Aditi compiler (Section 3.6) to translate the query into RL, the language of the DAP, and passes the translated query to the DAP to be executed. When the operation is complete, the DAP leaves its result in a temporary relation; the query shell looks up this relation and displays its contents to the user.

In the usual course of events, the temporary relation is then deleted. However, the user can request that the temporary relation be retained until the end of the current session by naming it, via a query such as

```
2 <- yankee_airlines(Airline) := hq(Airline, new_york).
```

This creates a new temporary relation `yankee_airlines` that can be accessed by later queries in this session. The new relation has arity 1 and its contents is the list of the airlines whose headquarters are in New York. Note that this query does not set up a rule; the content of `yankee_airlines` is not updated if the hq relation is modified.

Updates to permanent base relations are accomplished similarly, by prefacing a query with an atom referring to a permanent base relation and a += sign (for insertions) or a -= sign (for deletions). For example, the following query deletes all flights that either start or finish at snowed-in airports:

```
3 <- flight(From, To) -= flight(From, To),
   (snowed_in(From) ; snowed_in(To)).
```

The code of the query shell is an adaptation of the main loop of the NU-Prolog interpreter (which itself is written in NU-Prolog). The query shell's ability to record a history of previous queries, and to recall a given previous query, possibly edit it, and then execute it, derives from this source. For a full list of the capabilities of the query shell, see Harland et al. (1992*b*).

3.2 The SQL Query Shell

Aditi includes a prototype of a query shell that accepts SQL as input. This prototype is based on an SQL to NU-Prolog translator that was developed several years ago by Philip Dart at the University of Melbourne as an educational tool for our undergraduates. The translator implements most of the SQL standard, the main exceptions being null values and (for the time being) updates.

SQL queries do not make distinctions between base and derived relations. The query

```
4 <- select * from trip where not some
      (select * from flight where
      flight.origin = trip.origin and flight.destination =
                                                   trip.destination)
```

will elicit the same response from the SQL query shell as the query

```
5 <- trip(From, To, Trip), not flight(From, To).
```

will from the Aditi-Prolog query shell. Both ask Aditi to "list all trips in the database that are not direct flights."

The main problem with the SQL query shell is that SQL requires relations to be in first normal form, and relations in Aditi may violate this assumption by containing or computing non-atomic attribute values. Therefore, the SQL query shell can access only some Aditi relations (those in 1NF) without being confused.

3.3 The Graphical User Interface

The xqsh program presents a conventional graphical user interface (GUI) to the Aditi system. It is written in tcl using the tk toolkit (Ousterhout, 1994) and has a Motif-like look and feel. When it starts up, xqsh creates a window with three main parts: an area displaying the name of the current database, a text area, and a menu bar. Xqsh uses the text area to display a session with an Aditi-Prolog query shell; the result of each query appears in a separate popup window. The menu bar provides easy access to almost all of the other functions of Aditi. The menu bar has entries for

- creating new databases and deleting old ones
- switching the context to another (existing) database
- displaying lists of existing schemas and relations
- creating and deleting schemas and base relations
- displaying and modifying the contents of base relations

- displaying and modifying the permissions associated with base relations
- editing, compiling, and loading Aditi-Prolog programs

All these actions can be performed from a command line as well. Xqsh is useful because its prompts obviate the need to remember the order and type of what would otherwise be command line arguments, and because it supports browsing. The only significant functionality xqsh lacks is the ability to start and stop the query server. Since xqsh cannot function without a running query server, this functionality is available from a Unix shell command line only.

Xqsh is easily extensible and therefore specializable to the input and output requirements of particular applications. We have built a variant called xflight for our flights database (Harland and Ramamohanarao, 1992). This variant presents users with a form listing the various parameters of a trip to be planned (where from, where to, departure and arrival date and time limits, airline preferences, etc.), and asks them to fill in some or all of these parameters. When the users signal that they have filled in all the fields they want, xflight assembles the Aditi-Prolog query, executes it, interprets the results, and displays them on a map of the world.

3.4 Accessing Aditi from NU-Prolog Programs

An important feature of Aditi is the existence of a seamless interface between Aditi and the NU-Prolog programming language. This interface allows NU-Prolog programs to make calls to a predicate and receive the answers without being aware whether the predicate in question was defined in NU-Prolog or in Aditi. For example, consider the NU-Prolog code fragment below.

```
find_flights :-
    get_input(Origin, Dest, Date),
    trip(Origin, Dest, Date, Trip),
    cheap_enough(Trip),
    pretty_print(Trip).
```

The predicate `get_input` prompts the user for the origin and destination of a trip and the desired date of travel. The call to `trip` finds all trips (sequences of flights with an airline, flight number, and cost for each, with short stopovers, say less than four hours between them) between the two named locations starting on the named date. The call to `cheap_enough` acts as a filter, rejecting all trips above a certain price, while `pretty_print` prints the details of the trips that pass the `cheap_enough` filter.

The idea behind Aditi's NU-Prolog interface is the same as the idea behind SQL embedded in C or Cobol: Use a database language for database tasks and use a host language to take care of procedural tasks such as I/O. Therefore, in this example `get_input`, `cheap_enough`, and `pretty_print` will be NU-Prolog predicates, and `trip` will be an Aditi predicate.

To allow trip to be accessed from NU-Prolog programs, the application designer need only associate a flag with the predicate. When it sees this flag, the Aditi-Prolog compiler creates not only an object file for the DAP but also an interface file containing NU-Prolog code. NU-Prolog programs can then get transparent access to trip by linking this NU-Prolog code into themselves.

The automatically generated interface code is the system component responsible for resolving the impedance mismatch between tuple-at-a-time and set-at-a-time computation. The first call to trip will find all trips the user's parameters request, but it will return only the first one to NU-Prolog. Later solutions are returned only upon backtracking, which occurs either because cheap_enough has rejected a trip or because pretty_print has finished printing a trip and is looking for more.

The interface code also is responsible for selecting the appropriate variant of trip to call. As we discussed in Section 2.2, an Aditi-Prolog predicate may have more than one mode. For example, trip may be compiled in two modes: one expecting the first three arguments to be bound, and one expecting only the first two arguments to be bound. The interface code for trip will examine the instantiation state of all four arguments, and will call the first variant if the first three arguments are indeed all ground. Otherwise, it will call the second variant if it can. In either case, it will test the fourth argument of returned solutions (and in the second case, the third argument as well) to see whether they match the corresponding term in the query. If it can't call either variant because at least one of the first two arguments is not ground, it will report an error.

Some NU-Prolog programs may opt to store the answers to an Aditi query to allow them to be processed several times. In our example, the user may wish to see not only the cheap trips, but also the short trips and trips using only a particular airline. Rather than ask Aditi the same query three times, the program can store the answers internally in NU-Prolog as follows:

```
find_flights :-
    get_input(Origin, Dest, Date),
    dbQuery(trip(Origin, Dest, Date, Trip), Answers),
    process(Answers).
```

After the call to dbQuery, the variable Answers contains a pointer to a table containing the solutions of trip(Origin, Dest, Date, Trip). The program may access the elements of this table by the call readTuples(Answers, Trip); the interface predicate readTuples will successively instantiate Trip to one of the answers to the Aditi query. Thus, the problem of finding the cheapest flights, the shortest flights, and the flights on a given airline reduces to performing three searches of the table. These searches rely on cursors like those found in SQL environments; these cursors can be implicit (as above) or explicit. However, NU-Prolog is so close to Aditi-Prolog that many pieces of code are accepted by compilers for both languages, and so we expect very little need for the use of explicit cursors; they are provided more for completeness than anything else.

For more details on the Aditi/NU-Prolog interface, including the facilities for updating Aditi relations from NU-Prolog, see Harland et al. (1992*a*).

3.5 The Aditi Relational Language

RL is a simple procedural language whose primitive operations are needed for database implementation. These primitives include

- the standard relational algebra operations such as join, union, difference, select, and project
- extended relational algebra operations such as union-diff and btmerge, which perform two or more standard relational algebra operations at the same time but with only one scan of the input relations
- data movement operations such as append (union without checking for duplicates), copy (which copies a relation), and assign (which copies a *pointer* to a relation)
- operations concerned with data structure optimization such as presort-relation
- aggregate operations such as count, max, and min
- arithmetic and comparison operations on integers and floating-point numbers and the usual operations on boolean values
- operations to invoke a NU-Prolog interpreter (Section 4.1)
- operations to retrieve information from the data dictionary
- operations to create, delete, and clear relations
- operations to insert tuples into and delete tuples from relations
- operations to manage DAP threads (Section 4.1)
- operations to shut down the query server (and thus the DAP)

The control structures of RL are a similar mix of simplicity and sophistication: RL supports only go to statements (gotos), conditional branches, and procedure calls.

The procedures are the key to RL. An RL program has a procedure for every mode of every derived predicate in the Aditi-Prolog program from which it was compiled. As an example, consider the following Aditi-Prolog code:

```
?- flag(path, 2, context).
?- mode(path(b,f)).
path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
```

This code declares a derived predicate called `path` that expects to be called with its first argument ground, and directs the compiler to apply the optimization method known as the context transformation (Kemp et al., 1990). The body of the predicate defines `path` to be the transitive closure of the `edge` relation, just as the `trip` predicate is a kind of transitive closure of the `flight` relation. We have chosen path as the example because the additional details in the code produced for trip would needlessly clutter the explanation.

Given the four-line definition of path at the beginning of this section, the Aditi-Prolog compiler produces the following RL code. The only modification we have made for presentation was to add some white space.

```
procedure
path_2_1(init_path_2_1, output_path_2_1)
relation init_path_2_1, output_path_2_1;
{
  relation diff_mc_path_2_1_2_1, edge_4, final_mc_path_2_1_2_1;
  relation out_eq_1, out_eq_7, edge_9, output_mc_path_2_1_2_1;
  bool bool1;
  int size1;

  setbrel(final_mc_path_2_1_2_1, 2);
  settrel(diff_mc_path_2_1_2_1, 2);
  settrel(edge_4, 2);
  settrel(out_eq_1, 2);
  settrel(out_eq_7, 2);
  settrel(edge_9, 2);
  settrel(output_mc_path_2_1_2_1, 2);

  project(init_path_2_1,out_eq_1,``#(0,0),#(0,0)'');
  btmerge(final_mc_path_2_1_2_1,out_eq_1,diff_mc_path_2_1_2_1,0);
  setprel(edge_4, ``edge'', 2);
label1:
  join(diff_mc_path_2_1_2_1,edge_4,``#(1,0)=#(0,1)'','''',
     out_eq_7,``#(0,0),#(1,1)'');
  btmerge(final_mc_path_2_1_2_1,out_eq_7,diff_mc_path_2_1_2_1,0);
  cardinality(diff_mc_path_2_1_2_1, size1);
  gt(size1, 0, bool1);
  test(bool1, label1);
  flatten(final_mc_path_2_1_2_1,output_mc_path_2_1_2_1);
  setprel(edge_9, ``edge'', 2);
  join(output_mc_path_2_1_2_1,edge_9,``#(1,0)=#(0,1)'','''',
     output_path_2_1,``#(0,0),#(1,1)'');
  clear(diff_mc_path_2_1_2_1);
  clear(edge_4);
  clear(final_mc_path_2_1_2_1);
  clear(out_eq_1);
  clear(out_eq_7);
  clear(edge_9);
  clear(output_mc_path_2_1_2_1);
}
```

Note the (intentional) similarity to the "look and feel" of a C program. The name of the RL procedure is derived from the name of the Aditi-Prolog predicate it implements (path), its arity (2), and the number of the mode it implements (1). The names of some of the local variables have two copies of this suffix; one comes from the arity and mode of path, the other from the arity and mode of the auxiliary predicate mc_path_2_1 created by the compiler as part of the context transformation on path_2_1 (the "mc" stands for "magic context"). The RL code above is in fact compiled from code that looks like this:

```
?- mode(mc_path_2_1(b,f)).
mc_path_2_1(C, C).
mc_path_2_1(C, A) :- mc_path_2_1(C, B), edge(B, A).

?- mode(path(b,f)).
path(X, Y) :- mc_path_2_1(X, A), edge(A, Y).
```

This code exists only inside the compiler. Since the auxiliary predicate cannot be called from anywhere else, its RL code has been inlined into the RL code of path itself.

By convention, all RL procedures generated by the compiler 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_2_1 has one attribute while output_path_2_1 has two, because path has two arguments, only one of which is input in mode number 1. When path_2_1 is called, the init_path_2_1 relation must already be known, but path_2_1 is responsible for determining the contents of output_path_2_1 (in fact any tuples in output_path_2_1 at the time that path_2_1 is called will be overwritten and discarded).

The body of the procedure path_2_1 begins with the declaration of some local variables, and continues with the creation of empty relations, all of arity 2, to serve as the initial values of the relation-valued variables. The first of these initializations calls the procedure setbrel to give final_mc_path_2_1_2_1 a B-tree index whose key is the hash value of the entire tuple; the other temporary relations have no index. The call to setprel puts into edge_4 a pointer to the permanent base relation called "edge" of arity 2.

The arguments of many RL operations consist of the names of input relations and output relations, selection conditions, and projection specifications. Selection conditions and projection specifications contain occurrences of strings of the form "#(n,m)": these refer to attribute m of input relation n, where both attributes and input relations are numbered from zero.

The project operation therefore puts into out_eq_1 one tuple for each tuple in init_path_2_1, and both attributes of this tuple are taken from the only attribute of init_path_2_1. The btmerge operation then takes the out_eq_1 relation, inserts its tuples into final_mc_path_2_1_2_1, and puts any tuples that were not already in final_mc_path_2_1_2_1 into the relation diff_mc_path_2_1_2_1. Since final_mc_path_2_1_2_1 was initially empty, this will copy out_eq_1 into both final_mc_path_2_1_2_1 and diff_mc_path_2_1_2_1. This implements the first rule of mc_path_2_1.

The loop invariant being established is that, at the beginning of the Nth iteration, the second argument of final_mc_path_2_1_2_1 always holds the places reachable from the first argument by traversing N or fewer edges, and diff_mc_path_2_1_2_1 always holds the tuples that were inserted into final_mc_path_2_1_2_1 in the N-1th iteration (the initialization counts as iteration 0).

The loop starts with the label "label1": its body implements the second rule of mc_path_2_1 with differential evaluation. The join takes the mc_path_2_1 facts generated in the previous iteration and joins them to the edge relation as dictated by the conjunction mc_path_2_1(C, B), edge(B, A). The join condition states that #(1,0) and #(0,1), the first argument of the second input relation and the second argument of the first input relation, respectively, must be equal, since both correspond to the variable B. The result of the join is a relation with three attributes, representing the variables C, B, and A. Since the head of the rule contains only C and A, the result is projected onto C and A (#(0,0) and #(1,1), respectively) before it is assigned to relation out_eq_7 which thus contains tuples corresponding to the mc_path_2_1 facts we have discovered in this iteration. The empty string after the join condition is the argument of the join operation that would construct new values during the join itself (e.g., by performing arithmetic on attributes of the join result) for use in the projection specifications; this is not needed in this example (Section 4.3).

The btmerge operation scans the out_eq_7 relation, looking up each tuple in the final_mc_path_2_1_2_1 relation, and using the indexing on that relation. If the tuple already exists in final_mc_path_2_1_2_1, it does nothing more; otherwise it inserts the tuple into both final_mc_path_2_1_2_1 and diff_mc_path_2_1_2_1. This maintains the invariant and completes the processing of the second rule of mc_path_2_1.

The next three lines find out whether the loop should continue. The first puts the cardinality of the difference relation into the integer variable size1. The second sets bool1 to true or false, depending on whether size1 is greater than zero. If it is, then the iteration just finished has discovered new mc_path_2_1 facts and thus we must go back to the start of the loop at label1 to try to use these new facts to generate even more facts; otherwise, we exit the loop.

The remaining code is executed only after loop exit: it implements the only rule of the transformed path predicate. The flatten operation copies the B-tree relation final_mc_path_2_1_2_1 into the flat (non-indexed) relation output_mc_path_2_1_2_1. Then the join operation implements the conjunction mc_path_2_1(X, A), edge(A, Y), the join condition requiring the equality of the two A's, and the project specification putting the values of X and Y into the relation `output_path_2_1`. The remainder of the procedure releases the space occupied by the temporary relations.

The call to setprel between the flatten and the join is redundant because `edge_4` already has a pointer to the edge relation. Other possible improvements include clearing `out_eq_1`, `out_eq_7` and `diff_mc_path_2_1_2_1` immediately after their last uses. We are working to incorporate these optimizations into the compiler.

3.6 The Compiler

The compiler that turns programs written in Aditi-Prolog into RL is written in NU-Prolog (Thom and Zobel, 1988). 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. 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. The most important of these tasks is the selection of an appropriate sideways information-passing strategy or sip (Ullman, 1985) for each rule in the input Aditi-Prolog program.

To find a sip the compiler needs to know the valid modes of the relations appearing in the rules and, if they are base relations, it needs to know their indexing, too. The three sources of this information are the compiler itself for built-in relations (such as $=$, $\leq$, $<$, etc.), the input file which must declare the modes of the predicates it defines, and the data dictionaries of Aditi databases.

By default, the compiler assumes that all predicates defined outside the file being compiled are permanent relations or registered derived relations in the current database, whose pathname is specified by the ADITIBASE environment variable. The input file can override this assumption by using a declaration of the form `?- db(`*pathname*`)`. One can also arrange for access to databases other than the main one. To do so, the input file must include a declaration associating a handle with the pathname of that database, and then use another declaration to associate a local name with a permanent relation in that database. The declarations

```
?- database(db2, ''/users/aditi/example'').
?- import(db2, flight, 3, flight2).
```

together, which make calls to `flight2` in this file, actually refer to the `flight` relation of arity 3 in the database whose path is `''/users/aditi/example''`.

Once the legitimate modes of each relation are known, the compiler can proceed with the creation of sips. Generally, different orderings of body literals are appropriate for different modes of the predicate involved, so the compiler creates sips separately for each mode of the predicate. Every rule needs its own sip. The chief component in the creation of this sip is the ordering of the literals in the bodies of clauses. The compiler does this by a kind of insertion sort. From all the remaining uninserted literals in the rules, by default the compiler chooses the leftmost literal that has a valid mode given the variables bound by the already inserted literals. This selection rule can be changed using the flags `maxboundsips` and `minfreesips`, which cause the compiler to select the literal with the most bound arguments and the literal with the fewest free arguments, respectively. These are both reasonable heuristics for reducing the size of intermediate relations.

Users who require precise control over the chosen sip can use the default left-to-right sip strategy together with specialized versions of rules for each mode. Consider the definition

```
?- flag(path, 2, context).
?- mode(path(b,f)).
?- mode(path(f,b)).
path(X, Y) :- edge(X, Y).
path(X, Y) :- mode(b,f) :: edge(X, Z), path(Z, Y).
path(X, Y) :- mode(f,b) :: path(Z, Y), edge(X, Z).
```

This code is a modification of the `path` predicate of the previous section. It declares an extra mode, and it tells the compiler that in the recursive rule it should call the edge predicate first if path is called in mode `bf` but that it should call the edge predicate last if path is called in mode `fb`. These instructions establish a sip that makes the context transformation applicable in both modes of `path`. However, moded bodies are not always necessary for this: the `maxboundsips` and `minfreesips` flags establish the same sip. Moded bodies do allow different modes to have completely different rule bodies, not just ordering variations on the same rule body; some applications can put this capability to very good use.

Once the assignment of sips to rule bodies has been completed, every call in the body has a mode number associated with it, and the compiler from then on views each mode of each predicate as a separate procedure (the mode number becomes part of the procedure name, as we have seen in Section 3.5).

In the HDS to HDS level the compiler implements the optimizations that are defined in terms of source-to-source transformations. The optimizations that fit here include magic set transformation (Bancilhon et al., 1986; Beeri and Ramakrishnan, 1987), supplementary magic set transformation (Sacca and Zaniolo, 1987), counting set transformation (Bancilhon et al., 1986; Sacca and Zaniolo, 1986), constraint propagation (Kemp et al., 1989; Mumick et al., 1990), and context transformations for linear rules (Kemp et al., 1990). Currently, the compiler performs the magic, supplementary magic, and context transformations.

To help guide the implementation of these transformations, the compiler uses the predicate call graph, a graph that has one node for each mode of each relation, and whose edges represent caller-callee relationships. The strongly connected components (SCCs) of this graph represent sets of mutually recursive predicates. The call graph imposes a partial order on these SCCs, with predicates in higher SCCs calling those in lower SCCs but not vice versa.

When the compiler creates magic or context rules for a given predicate, by default the rules it creates build magic or context sets that reflect calls to the predicate only from other predicates in the same SCC. Each call to the predicate from higher SCCs results in the magic- or context-transformed predicates being computed anew, according to the Aditi doctrine of separate compilation of procedures wherever possible.

Unfortunately, this approach can be very inefficient when higher SCCs call the magic or context transformed predicate many times, especially if the input sets they specify overlap. To solve this problem, Aditi-Prolog allows users to declare that certain groups of predicates should be compiled together, and that all calls from a group to a predicate in that group should contribute to the same magic or context set for that predicate.

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. For mutually recursive predicates the compiler generates a single procedure containing one big iteration that computes values for all these predicates, and an interface procedure for each predicate involved. The interface procedures 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 default compilation approach is a semi-naive or differential evaluation (Balbin and Ramamohanarao, 1987), but naive evaluation, magic-set interpreter evaluation (Port et al., 1990), and predicate semi-naive evaluation (Ramakrishnan et al., 1990) are also available. This is also where the alternating fixpoint evaluation method (Kemp et al., 1991) required for non-stratified programs will fit in.

The magic set interpreter computes magic sets for the called relations while it is performing joins in the calling rule body. In the presence of negation within modules, the magic set transformation can turn stratified programs into non-stratified programs that cannot be evaluated with the present compiler.

Predicate semi-naive evaluation changes the way the iterations of mutually recursive predicates are handled. In normal semi-naive evaluation, the code of each predicate uses facts generated in the previous iteration; in predicate semi-naive evaluation, the code of a predicate can also use facts generated by the code of other predicates in the same iteration. This usually increases the efficiency of evaluation by reducing the number of iterations required to reach a fixpoint.

HDS to HDS optimizations come mainly from the logic programming/deductive database literature. LDS to LDS optimizations come from the programming language and relational database literature; they include such techniques as common subexpression elimination, loop invariant removal, early reuse of storage, moving selections before joins, and the intelligent exploitation of any available indexing. So far we have implemented only a few such optimizations.

The third main stage, LDS to RL, is necessary for two reasons. First, LDS has if-then-else- and while loops, whereas RL has labels and gotos. 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. (These differences are caused by the fact that the aim of LDS is easy optimization while the aim of RL is easy assembly and fast execution.) So the LDS to RL translator includes a peephole optimizer that can convert a sequence of LDS operations into a single RL instruction.

The output of the compiler is human readable RL code (see the example in the previous section). 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 new features of Aditi before the compiler modifications needed to make use of them have been completed. 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. The Servers of Aditi

Sections 4.1 through 4.3 describe Database Access Processes, Query Servers, and Relational Algebra Processes, respectively. Sections 4.4 and 4.5 describe the common code base on which all three types of processes are built.

4.1 Database Access Process

Users must go through DAPs to gain access to Aditi; the DAPs have the responsibility of enforcing the security restrictions (if any) on access to shared databases. DAPs are trusted processes; they always check whether the user has the authority to carry out the operation he or she requests the DAP to perform. A front-end process (such as the query shell) can get access to Aditi in one of two ways. It can create a child DAP process and set up a pipe to it, or it can have a copy of the DAP built in. Though the latter method is faster, for obvious reasons it is confined to application programs that are themselves trusted. An example of a trusted application is the NU-Prolog interpreter. Because a NU-Prolog program cannot compromise the internal data structures of the NU-Prolog interpreter, all NU-Prolog programs are trusted as well (Figure 1).

After startup and the completion of the obligatory privilege check, DAPs go into a loop, waiting for commands and executing them. These commands are not in Aditi-Prolog but in a binary form of RL; DAPs contain an interpreter for this language. Each DAP has a table containing the names and the codes of the RL procedures it knows. At startup, this table will be empty, but by looking up the data dictionary, the DAP can on demand retrieve the RL procedures of the database's derived relations.

To execute a query on a single predicate whose arguments are all distinct free variables, the front-end process tells the DAP to execute the corresponding RL procedure in its procedure table if the predicate being accessed is a derived relation; otherwise it tells the DAP to execute the built-in setprel command with the name of the base relation as an argument. To execute any query more complex than this, the front-end process must compile the query into an RL procedure, load that RL procedure into the DAP, and tell the DAP to execute that procedure. The DAP will leave the results of queries in temporary relations; the front-end can then ask the DAP for the contents of these relations and may use them in later queries (Section 3.1). If the result of the query is to be used as a set of tuples to be inserted into or deleted from a relation, the front-end will command the DAP to do that as well.

An RL procedure consists of a sequence of operations. The DAP interpreter executes some operations itself and sends others to the QS for assignment to some RAP. Operations that can require large amounts of computation, such as joins and unions, are performed by RAPs; operations with small time bounds, such as determining a relation's cardinality, creating a relation, or inserting a tuple into a relation, are performed directly by the DAP.

DAPs also execute one other kind of operation: those that are simply not suited for Aditi's usual evaluation algorithm. For example, when Aditi evaluates predicates for list manipulation such as list reverse and append in a bottom-up set-at-a-time manner, each iteration of the RL code involves several relational algebra operations, several context switches, and several buffer cache accesses just to add or remove one element at the front of a list. To avoid all this overhead, we have made top-down tuple-at-a-time computation available to Aditi users by embedding a NU-Prolog interpreter in each DAP.

When the compiler sees a predicate flagged as a top-down predicate in the Aditi-Prolog source file, it copies the predicate to a NU-Prolog source file and compiles it using the NU-Prolog interpreter. Calls to this predicate are compiled into an RL instruction that invokes this NU-Prolog object code in the DAP's NU-Prolog interpreter. Note the predicate must be acceptable to both compilers by which it is processed, Aditi-Prolog and NU-Prolog—this is not difficult due to the close relationship between the two languages.

Apart from top-down predicates, most predicates spend most of their time in the RAPs. Normally, whenever the DAP sends a job to a RAP through the QS, it will wait for the RAP to report the completion of the job. However, the compiler can generate code in which sending a job to a RAP and waiting for a completion report are two separate actions, in between which the DAP can continue running and perform work in parallel with a RAP. Because the DAP can continue to issue other jobs to other RAPs, several RAPs can work for the same DAP at the same time.

The two main types of parallelism in Aditi-Prolog programs are parallelism between atoms inside a clause, and parallelism between clauses. Parallelism between atoms in a clause is possible only if neither atom needs a value generated by the other. Parallelism between clauses is restricted by the nature of bottom-up evaluation, which must apply nonrecursive clauses before it goes into a loop applying the recursive clauses. Therefore, parallelism is possible only among the nonrecursive clauses and among the recursive clauses, not between a nonrecursive clause and a recursive clause. Many predicates have only one recursive clause. However, if this clause is non-linear (i.e., it has two or more recursive calls) then differential evaluation and the magic set transformation will both effectively split this clause into two or more clauses that can be evaluated in parallel.

The DAP executes every RL operation in the context of a thread. Sequential RL code only has one thread. Parallel code creates and destroys threads as necessary. We have developed two compilation schemes for managing thread activity (Leask et al., 1991). One is based on the fork-join approach and the other on dataflow methods. The fork-join approach is very simple: whenever the compiler knows that two or more operations or operation sequences are independent, it creates a thread for each operation sequence. The main thread launches all these threads simultaneously and then waits for all of them to complete before continuing.

In theory, the compiler will generate one thread for each nonrecursive clause, with each of those threads launching separate threads for separate atoms, and it will generate a similar thread structure inside the loop. In practice, the thread structure will be less complicated because independent atoms inside clauses are somewhat rare, and because many important predicates have only one non-recursive clause and/or only one recursive clause. However, as long as the predicate has *some* independent operations, the fork-join scheme will exploit it.

The weakness of the fork-join approach is its handling of loops. It cannot start the next iteration of a loop until the threads of all recursive clauses have completed, even if the threads that have completed so far have produced the relations needed by some of the threads to start their part of the next iteration. This leads to a loss of concurrency. Our other compilation scheme therefore controls synchronization between threads by dataflow methods: A thread can execute its next operation if the operations that produce its input relations have completed.

Our experiments have proven that the dataflow method can yield greater concurrency and better performance than fork-join without increasing run-time overhead, but only if a set of mutually recursive predicates have some recursive rules independent of each other, which is quite rare. The cost is increased complexity in the RL interpreter and in the Aditi-Prolog compiler. In fact, while the RL interpreter in the DAP supports both schemes (it had to for us to run our experiments), the compiler currently generates only fork-join code.

Note that neither scheme requires the DAP itself to be parallel. All of the parallelism comes from the parallel operation of several RAPs. The DAP does maintain several thread contexts, but it executes in only one context at a time; it switches to another thread only when its current thread exits or blocks waiting for a reply from a RAP. Since the operations native to the DAP are all much shorter than a typical RAP operation, this is a very effective simulation of time-slicing.

4.2 Query Server

The Query Server (QS) is the central management process of Aditi; it must be running for Aditi to be available to users. At startup, it is responsible for initializing the entire database system according to a specified global configuration file and starting the appropriate number of RAPs. During normal operation, its main task is to keep track of the state of each RAP and to allocate tasks to free RAPs. The QS listens on its message queue waiting for various requests and either performs them immediately or queues them until they can be performed. Requests can come from DAPs and RAPs. A DAP can make requests to login to Aditi, to have a task executed, to get the status of the system, and to logout from Aditi. It can also send the QS a message to shut down the system if its user is authorized to do so. A RAP can tell the QS that it has finished an assigned task and is ready for another task, or it can inform the QS that it is aborting due to some fatal error.

By controlling the number of running RAPs and DAPs and keeping usage statistics for each user, the QS can perform load management for Aditi. Controlling the number of RAPs and DAPs allows the QS to prevent Aditi from overloading the host system. Recording usage statistics means that the QS can prevent individual users from overloading Aditi itself.

The QS can control the number of DAPs by refusing to login new DAPs when the load on the system is too high. Since DAPs place relatively small loads on the system, such action should be quite rare. Control over the number of RAPs is more critical; relational algebra operations are the most expensive operations performed in a database system and can easily overload the host machine. The QS prevents this by not passing DAP requests onto the RAPs when the load on the machine is dangerously high, and by not creating more RAPs than the machine can support. This way many DAPs can be running, providing service to many users, but the overall number of jobs being performed is strictly controlled.

4.3 Relational Algebra Process

Relational Algebra Processes are the workhorses of Aditi; they execute the most expensive operations in RL. They are all relational algebra operations, either standard ones (e.g., join, union, difference, select, project) or combinations such as union-diff and btmerge (Section 3.5). RAPs spend their lifetime waiting for a job from the QS, executing the job, and sending the result to the requesting DAP while notifying the QS of their availability. A RAP exits only when the QS commands it to shut down or it encounters a fatal error.

The message to a RAP specifies the name of the operation to be performed, the input relations, select conditions for these input relations, any data specific to the operation (e.g., join attributes), and a list of output relations along with projection information that specifies how each tuple in those relations is to be constructed from the result of the operation. All relational algebra operations can therefore be preceded by select operations and followed by project operations. The select conditions on the input relations often allow the RAP to confine its attention to the relevant parts of the input relations without incurring the overhead of creating temporary relations to hold those parts. In the same spirit, the projection information allows the RAP to allocate storage only for the part of the output that is actually required; the fact that more than one output relation can be specified (presumably each with a different projection) allows the RAP to avoid the overhead of scanning the result of the operation several times. A secondary benefit of pre-select and post-project is a reduction of the communication overhead between the DAP and the RAP (this is also true for combination operations like btmerge).

Depending on the size and structure of the argument relations, the RAP can choose from several algorithms to carry out its various operations. The operation whose performance is most important is the join. Since no single join method works optimally in all situations, Aditi currently supports four join algorithms: sort/merge join, hash join, nested block join, and indexed join.

Sort/merge join, like other sort-based Aditi operations, uses the external merge-sort algorithm described by Knuth (1973). To make sure that the overhead of extracting sort keys from a tuple is paid only once, rather than every time the tuple appears in a comparison, our implementation uses prepended sort keys (Linderman, 1984). Since the schema format supports relations with prepended keys, our sort routine will leave the sort keys in place if the calling procedure indicates it wishes to use them. Thus, relations can have their keys extracted and be sorted just once instead of once per operation; such presorting can substantially reduce query time.

Our sort routine can write its results to a temporary relation or pass them back on demand to the calling RAP operation. A RAP executing a sort/merge join will sort each input relation on the attributes being joined, and then scan the results looking for matching tuple pairs. The sort routine will help the join algorithm in two ways. First, it removes duplicate tuples as soon as it discovers them. Duplicate elimination has proved essential for recursive queries because it restricts the exponential growth of the sizes of temporary files. Each iteration of a bottom-up computation can double the number of tuple copies in a relation (and more than double the time required to process the relation if it outgrows memory). Second, the merge-sort routine performs the final merge pass on demand just as the RAP requests the next tuple in its result; the final merge passes on the two input relations are interleaved with the scanning pass of the join algorithm itself. This arrangement reduces the number of passes over the data and the attendant I/O overhead. However, the sort/merge code still starts with a check to see whether either relation is already sorted on the join attributes; correctly sorted files are used as they are.

The sort/merge join sorts to bring tuples with the same join attributes together. The *hash join* algorithm does this clustering by splitting each input relation into partitions based on the hash value of the join attributes of each tuple. It then joins pairs of partitions with the same hash values in a nested loop fashion.

The hash join and the sort/join methods are applicable in the same situations. The hash join will generally perform better, exceptions mostly caused by its greater sensitivity to data skew and its inability to always remove duplicates from the input relations. This is because not all of a partition will be in memory when a tuple is inserted into that partition, and hence the tuple cannot be compared against all tuples in that partition.

Although we are considering various heuristics that will allow us to choose dynamically between sort/merge join and hash join for joining large relations, currently the decision is made statically in the query server's configuration file. The configuration file also gives the size below which a relation is considered small: if at least one input relation to a join is small, then the RAP uses one of its other join methods.

The first of these methods, *nested block join*, requires that one of the relations be small enough to fit into memory. It loads the tuples from the small relation into a hash table in memory, and then scans the tuples in the large relation sequentially. As the scan reaches a tuple, the algorithm hashes its join attributes and looks in the hash table for tuples to join with this tuple. Because this algorithm examines the data in each relation only once, it is about four times faster than either sort/merge join or hash join. This speedup factor means that splitting up the small relation into memory-sized chunks and using nested block join on each chunk is worthwhile whenever the small relation is less than four times the buffer size. It is the size of this memory buffer that is governed by the configuration file; we avoid computing this size automatically to allow us to switch easily between the various join methods for performance testing.

The last join method supported by Aditi is *indexed join*. For each tuple in the small relation, the algorithm performs an indexed lookup on the other relation to retrieve all the tuples that have matching attribute values, and then joins them with the tuple from the small relation. This is the preferred method when one of the input relations only has a small number of tuples, and the other input relation has an index on the attributes required for the join. We do not use it when both relations are large because this would require a large number of lookups with bad locality. In such cases, the better locality of sort-merge join yields better performance, even though it accesses more tuples (unless the ratio between the size of the larger relation and the threshold is greater than the ratio of I/O costs of indexed lookups and sequential accesses).

Currently, only permanent relations can be indexed on attributes and so only joins with permanent relations will use this method. (The sort key of B-tree indexed temporary files is the hash value of the entire tuple, since this is needed for duplicate elimination.) Joins involving only temporary relations will use only the other join methods. This is not too much of a problem because temporary relations tend to be relatively small, and joins involving large relations are very likely to have at least one input that is a permanent relation. Our tests show quite clearly the importance of indexing permanent relations to allow the use of this join method. Appropriate indexing accelerated some of our tests by factors of up to 100 (Section 5).

Currently, each relational algebra instruction is carried out on a single RAP, but there is no fundamental reason that multiple RAPs cannot cooperate on a single task. We are working on schemes to enable multiple RAPs to co-operate in sorting a single relation in parallel, and on parallel hash based join algorithms (Nakayama et al., 1988; Tham, 1993).

4.4 The Database Library

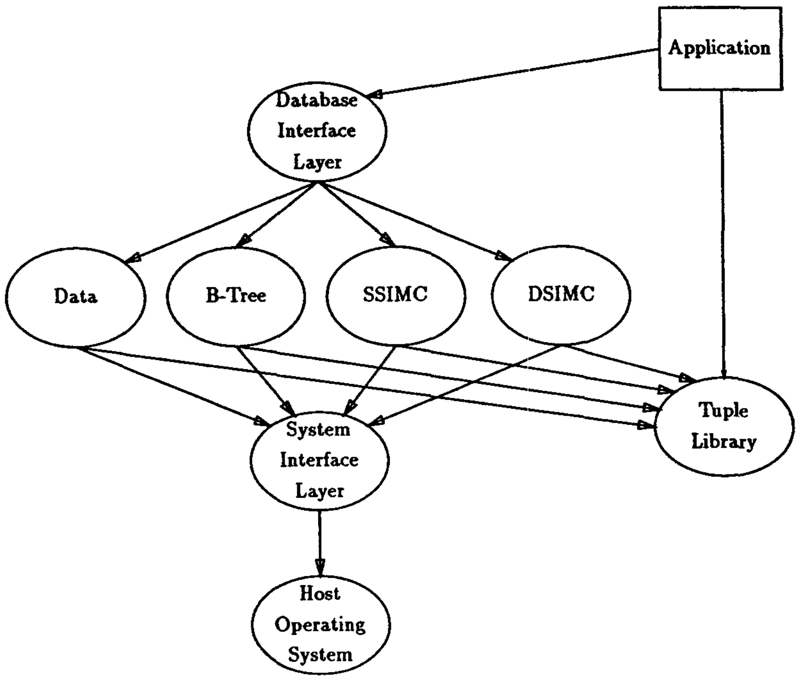
An Aditi database is a set of relations implemented as files and subdirectories, and stored in one directory in the Unix file system. Every database has a data dictionary relation containing information about all its schemas and relations. We designed Aditi so that the implementations of RL operations are as independent as possible of the structure of the relations they are manipulating. Although operations can find out this information, they are mostly content to leave low-level relation handling to the database library. Together with the tuple library, the database library forms the lowest level of Aditi; Figure 2 shows how they fit together.

4.4.1 The Indexing Layer. The database library provides routines for creating and deleting temporary and permanent relations; inserting, deleting, and retrieving tuples from relations; and tuple-level and relation-level locking. The task of the top layer in the database library, the interface layer, is to hide the differences between the implementations of these operations by the various indexing methods supported by Aditi. The database interface layer therefore acts as a switch; for the most common operations, the real work is done by code specific to an indexing method. This code forms the indexing layer, the layer below the interface layer.

4.4.2 The System Interface Layer. The lowest layer of Aditi is the system interface layer, which provides the rest of Aditi with a set of low-level services including message passing, locking, and disk buffering and caching. The two reasons for the existence of the system interface layer are: our desire for OS independence in the rest of Aditi to make porting easier; and the need to bypass inefficient OS mechanisms whenever possible.

The current version of the system interface layer runs on most Unix implementations that support the IPC facilities of System V, in particular System V shared memory. Some type of shared memory is essential for good performance, since it allows Aditi's server processes to communicate among themselves without incurring the overhead of system calls; they use the message-passing routines supplied by the system interface layer instead. These routines pass messages in shared memory

Figure 2. Structure of database and tuple libraries



protected by spinlocks, and invoke System V semaphore operations only when a process needs to block waiting for an event. Part of the process of porting Aditi to new hardware involves writing the spinlock routines in assembler.

To gain control over what parts of which relations reside in memory Aditi, like many other DBMSs, implements its own buffer cache. In fact, Aditi supports two kinds of buffer cache: a single shared buffer cache stored in shared memory, and private buffer caches stored in the private virtual memory of individual server processes.

The shared buffer cache is for caching shared database pages (pages from the database dictionary and from permanent base relations). Access and modification of these pages is synchronized between competing processes to ensure the pages are not corrupted. As a result, processes may be delayed, waiting for access to pages containing the data they want, or waiting for access to free pages for their own use.

The purpose of the private buffer caches is to help reduce these contention problems. They were specifically designed to hold temporary relations during query processing. Since a deductive database query produces many intermediate relations which are only visible to that query, accessing all these relations through the shared buffer cache would significantly increase process conflicts without deriving any benefits from data sharing. The compiler ensures that only one process will ever be writing to a given temporary relation and that no other processes will access this relation while it is being written. However, several processes may read a single temporary relation simultaneously; this can occur if parallel query processing is being used.

The problem with this scheme is that if any process other than the creator needs access to the temporary relation, that process cannot read it from the creator's private cache. Instead, the creator must flush the relation out of its cache onto disk, where the other process can see it. For a short-lived temporary relation, the benefits of reduced contention may be squandered in the extra disk traffic. Profiling tests have shown that the creating and unlinking of the disk file for a temporary relation can often be the most significant part of the overall CPU and I/O costs associated with a temporary relation. Much of this cost is due to the pathname-to-inode translation that takes place within the kernel whenever a file is created or unlinked.

To alleviate these problems, Aditi implements what we term "in-core" temporary relations. Aditi stores the names of in-core relations in a table in shared memory, and allocates pages to such relations from the shared buffer cache. Once allocated, such pages remain locked in memory: they are never written to disk and are only released when the in-core relation is deleted. Due to size limits on shared memory, there are limits on the number of in-core relations that can exist at any one time; on the number of pages that an in-core relation can have in memory; and on the total number of buffer cache pages that can hold data from in-core relations (we typically set the last limit at about 30% of the cache size). Pages in excess of these limits are stored on disk. The in-core relation table keeps track of the number of such overflow pages, as well as the name of the file in which they are stored and their location.

This arrangement solves both problems: pages in the shared buffer cache do not have to be flushed to disk to become visible to other processes, and disk I/O is avoided for small and short-lived temporary relations by delaying disk I/O and even the creation of the disk file until overflow actually occurs. Since most temporary relations are small and short-lived, this is an important optimization. Our data show that in-core temporary relations typically speed up query processing by a factor of four, the main benefits coming from the avoidance of disk file creates and unlinks. The query processor therefore creates all temporary relations in-core by default. If ever a temporary file cannot be created in-core due to the maximum in-core file limit being reached, the query processor will create it as a normal disk file.

In-core temporary relations are implemented at the lowest level as a storage type. The higher levels need only specify a flag at create time to say whether a temporary relation should be in-core or not. Thus we can create unindexed files, btree files, ssimc files, and dsimc files in-core with no change to any of the indexing code.

At present, Aditi has no transaction mechanism. We are currently modifying the code of the buffer caches to support the ARIES model of write-ahead logging (Mohan et al., 1992). We are also looking at higher level issues related to transactions (e.g., 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 computation resources. We would prefer a hybrid solution that minimizes these problems.

4.4.3 Security. Aditi has a very simple security system that provides minimal protection of a user's data. The security system recognizes only two levels of privilege: Aditi superusers (DBAs) and other registered Aditi users. The system controls who can read, modify, or delete a relation. When a relation is created, only the creator has access to it. The creator can then modify the relation's permissions in the following four ways:

- The creator can allow all users to read the data in the relation.
- The creator can allow all users to insert data into, delete data from, or modify data in the relation. This implies the ability to read the relation.
- The creator can allow DBAs (Aditi superusers) to delete the relation.
- The creator can allow any user (including DBAs) to delete the relation.

The creator of a relation always remains the only user who can change privileges on a relation. Any of the above privileges can be revoked at any time. Revoking read privilege automatically revokes write privilege, and revoking superuser delete privilege automatically revokes public delete privilege.

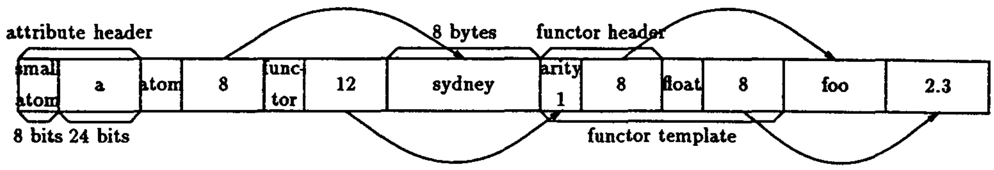
Currently, Aditi has no way of restricting the set of users who can create relations in a database. As long as a user has permission to attach to an Aditi server, that user can create a relation in any database visible from that server. This, of course, will change as Aditi matures.

4.5 The Tuple Library

The tuple library hides the differences between tuple formats from the rest of Aditi, which must invoke routines or macros supplied by the library whenever a look inside tuples is needed. Since all tuples in a relation must have the same format, the tuple library finds the proper algorithms to use for a relation by looking up the relation's schema in the data dictionary.

The tuple library currently provides three kinds of tuple structures. The first tuple format is simply a block of memory with a length field; this is intended for the storage of RL programs in Aditi relations. The second format consists of fixed length records with each attribute having a predefined type and maximum size; this is intended for relations of the traditional relational database type. The third format consists of variable length records in which each field may be of arbitrary size, and may include function symbols nested arbitrarily deep; this is intended for the storage of the structured terms one may find in Aditi-Prolog programs. None of the three formats supports the storage of variables inside tuples.

Figure 3. Layout of the tuple <a, sydney, foo(2.3)>



The first two tuple types are only in experimental use now, since Aditi currently stores RL code in Unix files and it does not yet have a type system that can impose maximum sizes on attributes. Therefore, all Aditi applications so far use the third tuple type, which we call "flexible." This structure allows arbitrary ground Prolog terms (e.g., lists with nested function symbols) to be stored directly in a relation and then manipulated by code written in RL. RL operations can project subterms and construct new terms as well as test the values of terms and subterms.

To support fast access to individual attributes, the flexible tuple format has a fixed sized header for each attribute, and it stores these headers contiguously. This header is a 32 bit word, of which the top 8 bits are for type information and the remaining 24 bits are either for the data, if they fit, or for an offset pointing to where the data are stored in the tuple (since this offset is a byte offset, tuples can be up to 16 Mb in size). Small signed and unsigned integers, very short strings, and atoms with very short names fit directly in the header; large integers, floating point numbers, long strings, atoms with long names, and all structured terms do not.

For example, consider the tuple <a, sydney, foo(2.3)> whose storage map appears in Figure 3. When creating this tuple, Aditi first allocates space for 3 32-bit headers. Since the atom "a" fits into 24 bits, it tags the first attribute as a "small atom" and inserts "a" in its data field. Since the atom "sydney" does not fit into 24 bits, Aditi pads it out to the nearest word size, in this case to eight bytes, and puts it at the current end of the tuple. Aditi then tags the second attribute as a plain "atom," computes the distance from the start of the second attribute position (at offset 4) to where the actual value is stored (at offset 12) and inserts 8 as the relative offset in the data field.

We chose the attribute position rather than the start of the tuple as the base for the offset to allow attributes to be passed around without needing to know from which tuple they came. The padding is needed to ensure that all headers are always word aligned, which in turn is necessary on most machines to allow numeric values such as integers to be used directly, without having to first move them to a word-aligned position.

When Aditi inserts the third attribute, which is a functor with sub-terms, into the tuple, it tags the third header as a functor and stores a relative offset of 12 in the data area (the tuple is now 20 bytes long, and the third attribute itself starts at offset 8). It then constructs a functor template at the current end of the tuple. The functor template consists of a functor header followed by an array of attribute headers, one for each of the subterms of the functor. In this case, the functor has arity 1, so there is only one attribute header.

The functor header is a 32 bit quantity with the top 8 bits containing the arity and the bottom 24 bits containing an offset to the functor name, which is stored directly after the functor template. (Since the length of the template is implicit in the arity, this offset is redundant, but its presence saves us from having to calculate it each time.) In this example, the relative offset to the functor name is 8: "foo" is stored after the functor template padded out to four bytes.

Aditi inserts a subterm into a functor template exactly as it performs the insertion of an attribute into a tuple. This is possible because arrays of headers for the subterms of a functor look exactly the same as arrays of headers for the attributes of a tuple. Therefore the insertion of the value "2.3" tags foo's only attribute header as a "floating point number" and puts the bits representing the FP number 2.3 at the current end of the tuple and stores the relative offset (8) in the data field of the attribute header.

Overall, the tuple is 36 bytes in length, but the tuple library does not know this until the tuple has been completely constructed. Accordingly, the tuple building routines always initially allocate a small amount of memory, and check how much space is left as they insert new values. If they run out, they allocate a piece of memory twice as large as before, copy the tuple to the new space and then insert the value. This copying is needed to keep tuples contiguous, but it can become expensive, especially if the tuple exceeds its current space allowance several times. To avoid this repeated copying, layers above the Aditi tuple library can set the size of the initial space allocation or even supply their own space if they know how large the tuple will be.

5. Performance

In this section we report on two sets of experiments. All tests were performed on a Silicon Graphics 4D/340S running IRIX 4.0.1 with four 33 MHz R3000 processors and 64 megabytes of memory. The tests were carried out in multiuser mode with other users present and active, but with the system load average below 2 most of the time. The test database was stored on a striped filesystem, track-interleaved across two SGI-supplied disk drives connected via two separate 4.0 Mb/s synchronous SCSI-1 interfaces. The drives have an average seek time of 11.9 ms, a single-track seek time of 2.5 ms, an average rotational latency of 6.25 ms (4800 rpm), and a maximum transfer rate of 2.3 Mb/s. The times we report are elapsed real times in seconds computed as the average of four or more runs. We report elapsed times instead of CPU times because they show how the system would appear to a user (CPU time measurements leave out context switches, disk I/O, and other similar effects).

The first set of experiments used the `path` predicate of Section 3.5 to compare the context transformed and magic set transformed programs against the original

Table 1. Results for path queries

Small query						
size	8	10	12	14	16	18
orig	4.9: 1.0	28.9: 1.0	246.9: 1.0	3069.8: 1.0		
magic	0.3: 16.0	0.6: 50.1	1.7: 143.8	7.3: 423.0	38.0: NA	289.1: NA
context	0.2: 20.1	0.4: 80.4	0.8: 329.2	2.2: 1398.6	7.8: NA	30.4: NA

Large query						
size	8	10	12	14	16	18
orig	5.5: 1.0	30.6: 1.0	257.6: 1.0	3058.0: 1.0		
magic	5.6: 1.0	17.8: 1.7	56.8: 4.5	168.3: 18.2	453.0: NA	1160.0: NA
context	4.4: 1.3	8.6: 3.5	24.0: 10.7	52.1: 58.7	107.8: NA	209.2: NA

program. Our test query was `?- t(X), path(X,Y)` with the base relation `t` supplying input values for the call to `path`.

Our experiments varied the contents of the relations `edge` and `t`. The `edge` relation contained full binary trees of various sizes (i.e., tuples of the form edge(i, $2i$) and edge(i, $2i+1$) for values of i ranging from 1 up to a maximum that depends on the size of the tree. Our six data sets used maximums of 255, 1023, 4095, 16383, 65535, and 262143, corresponding to tree depths of 8, 10, 12, 14, 16, and 18, respectively. Each version of the `edge` relation had a B-tree index on its first attribute.

We used two versions of the `t` relation. The first contained only the tuple "100." We label the results achieved with this version "small query." The other version contained 100 tuples randomly chosen from the middle three levels of the full binary tree of the relevant `edge` relation; we label the results achieved with this version "large query." We chose only trees of even depths for `edge` because the middle three levels move downward one level only when the total depth of the tree increases by two.

Table 1 reports our results. Rows correspond to optimization flags on the `path` relation, while columns correspond to the depths of the `edge` relation. Speedups are computed with respect to the untransformed program; they follow the times they refer to after a colon. One can make several observations based on the data in the table.

The number of tuples in the `t` relation makes virtually no difference for the untransformed program because it computes the entire `path` relation and then joins it with the `t` relation.

The magic set-transformed program always performs better than the untransformed program, and the context set-transformed program always performs better than the magic transformed program. The magic/original performance ratio climbs very rapidly as the size of the `edge` relation increases. The context/magic performance ratio also climbs but not as rapidly, reaching a maximum of 9.5 to 1 at depth 18 for the small query and a maximum of 5.5 to 1 at depth 18 for the large query. This illustrates that the context transformation performs relatively better when the input to the transformed predicate is more specific.

Database designers can use the gain in efficiency that can be achieved by the use of the magic and context transformations in two ways. First, they can pass it on to users in the form of better response times (e.g., 2.2 seconds vs. 51 minutes for the small query at depth 14, or subsecond responses at smaller depths). Second, they can maintain response times while increasing the size of relations. On the small query, the performance of the context transformed program at depth 18 is about the same as the performance of the untransformed program at a depth of about 10, or the performance of the magic transformed program at a depth of about 16.

The data in the table for depths 12 and 14, where the single tuple in the small query is in the middle three levels of the tree, show that multiplying the number of tuples in the input by 100 causes the time taken by the magic and context transformed program to go up only by a factor between 20 and 35. The reason for this is that part of the cost of query answering goes into the overheads of relational algebra operations (in the DAP, the RAP and interprocess communication). Since the large query causes the same number of relational algebra operations to be performed as a small query, the absolute cost of this overhead is the same in both cases. The cost of the overheads is a much higher proportion of the overall cost for the small query than for the large query.

The fact that figures grow faster in the small query table than in the large query table is at least partially due to the fact that the single tuple in the single query stays at one level in the tree while the hundred tuples in the large query move down in the tree.

Earlier papers on Aditi used a similar experiment but used different ways of picking tuples for the t relation; therefore those results are not strictly comparable with the results presented here. However, as a rough indicator our current results represent speedups of ten to 50 times over the results presented in previous papers on Aditi (Vaghani et al., 1991). These speedups come from improvements in the Aditi back-end and in the Aditi compiler, the main improvements being the use of indexes in joins, the use of btmerge to eliminate duplicates, and keeping temporary relations in main memory whenever possible. We expect these results to continue to improve in the future.

Our other test program is based on the first realistic application of Aditi, which is a database containing data about flights around the world (Harland and Ramamohanarao, 1992). Since we were unable to get real data in sufficient quantity, the information in this database, although derived from schedules published by airlines, is mainly the product of our imagination.

The main predicate of this database is the trip predicate. It finds flights which depart from the city named by the From argument and arrive at the city named by the To argument and which depart between the times Earliest and Latest on Ddate, waiting for no more than Stime hours at in-between stops. The rest of the arguments give the details: departure date and time, arrival date and time, a list with details on each flight segment, and the number of stopovers on the entire trip.

```
?- aditi(trip(b,b,b,b,b,b,f,f,f,f,f,f)).
?- flag(trip, 12, magic).

trip(From, To, Ddate, Earliest, Latest, Stime,
    date(D1,M1,Y1), Dtime, date(D2,M2,Y2), Atime,
    [flight(A, N, departure(From, date(D1,M1,Y1), Dtime),
        arrival(To, date(D2,M2,Y2), Atime))],
    0) :-

    flight_during(From, To, Ddate, Earliest, Latest,
        date(D1,M1,Y1), Dtime, date(D2,M2,Y2), Atime, A, N).

trip(From, To, Ddate, Earliest, Latest, Stime,
    Ddate2, Dtime, date(D2,M2,Y2), Atime1,
    flight(A, N, departure(Stop, date(D1,M1,Y1), Dtime1),
        arrival(To, date(D2,M2,Y2), Atime1)).F,
    NewStopovers) :-

    feasible(From, Stop, To),
    trip(From, Stop, Ddate, Earliest, Latest, Stime,
        Ddate2, Dtime, Adate, Atime, F, Stopovers),
    NewEarliest is Atime + 100,
    NewLatest is Atime + Stime,
    flight_during(Stop, To, Adate, NewEarliest, NewLatest,
        date(D1, M1, Y1), Dtime1, date(D2, M2, Y2), Atime1, A, N),
     NewStopovers is Stopovers + 1.
```

The four queries all involve a hypothetical traveller called Phileas Fogg who wants to travel around the world, spending one week in each of several regions. The four queries differ in the constraints imposed on the tour.

- Tour 1 must visit Asia, Europe, North America, and the Pacific region.
- Tour 2 must visit Asia, Europe, and North America.
- Tour 3 must visit Europe, North America, and the Pacific region.
- Tour 4 must visit Europe and North America.

The tours must visit the named regions in the order in which they are given; all tours start and finish in Melbourne, Australia. The queries all follow the same pattern; the structure of the query for tour 4 is:

```
?- place(Dest1, europe), trip(melbourne, Dest1, ...),
    place(Dest2, north_america), trip(Dest1, Dest2, ...),
    trip(Dest2, melbourne, ...).
```

Table 2. Results for Phileas Fogg queries

Results for queries with daily schedule

Query	Data		Dsimc		Btree	
	Magic	Context	Magic	Context	Magic	Context
Tour1	381.1: 1.0	282.3: 1.3	20.5: 18.6	14.4: 26.5	17.5: 21.8	13.9: 27.4
Tour2	294.4: 1.0	232.3: 1.3	16.9: 17.4	11.7: 25.2	14.1: 20.9	11.0: 26.7
Tour3	360.2: 1.0	266.5: 1.3	18.0: 20.0	14.0: 25.7	15.4: 23.4	13.5: 26.6
Tour4	285.6: 1.0	211.1: 1.3	14.2: 20.0	11.7: 24.4	12.7: 22.5	10.5: 27.1

Results for queries with weekly schedule

Query	Data		Dsimc		Btree	
	Magic	Context	Magic	Context	Magic	Context
Tour1	30.3: 1.0	24.3: 1.2	28.6: 1.1	21.4: 1.4	27.9: 1.1	23.2: 1.3
Tour2	24.6: 1.0	19.4: 1.3	23.5: 1.0	16.6: 1.5	23.3: 1.1	18.1: 1.3
Tour3	28.2: 1.0	23.0: 1.2	25.3: 1.1	20.7: 1.4	26.1: 1.1	22.0: 1.3
Tour4	22.4: 1.0	17.8: 1.3	19.3: 1.2	15.1: 1.5	20.5: 1.1	16.8: 1.3

We have two implementations of the predicate `flight_during` that finds out whether there is a flight between a given pair of cities during a particular time period. One uses a daily schedule that associates the availability of flights with an absolute date; the other uses a weekly schedule that associates this information with days of the week, subject to seasonal restrictions. Airlines usually publish their schedules in the compact weekly format, but this format requires some processing before use.

We have tested all four queries with both daily and weekly schedules, with the predicate finding trips between cities compiled with the magic set optimization, and with the context transformation, and with the schedule relation being stored without indexing, with dynamic superimposed codeword indexing and with B-tree indexing. The keys used for indexing are the origin and destination cities together with the desired date of travel. We did not include data for the the trip-finding predicate when it is compiled without optimization because that predicate is allowed only with respect to queries that specify the starting-date argument, and therefore it cannot be evaluated bottom-up without first being transformed by a magic-like optimization. The test results appear in Table 2, whose speedups are computed with respect to the magic transformed program using no indexing.

Table 2 tells us several things. First, the context transformation consistently yields results 20% to 40% better than the magic set optimization. Second, the type of indexing has a significant impact only for the daily schedule, in which case the schedule relation contains 54,058 tuples.

The four queries have 18, 12, 57, and 38 answers, respectively. This is not apparent from the table for two reasons. First, the tours with more answers are those that visit fewer regions and thus call `trip` a smaller number of times. Second,

Table 3. Results for retrievals with partially-specified keys

Query	Btree	Dsimc	Data
From Melbourne	5.04	0.51	15.49
To Melbourne	16.19	0.37	14.90

the cost of the joins invoked by `trip` depends mostly on the sizes of the input relations and very little on the size of the output relation.

As one expects, accessing such a large relation without an index has a large penalty, ranging from about 17- to 24-fold. For these queries the trip predicate always specifies all three of the key arguments of the schedule relation, so B-tree indexing is as effective as it can be. The dsimc indexing yields slightly lower performance (by about 10% to 20%), mainly because dsimc uses the keys only to restrict its attention to a set of pages and cannot focus directly on the tuples of interest within those pages.

For the weekly schedule, in which the relation contains 1,044 tuples, most of the time is spent in computation, not retrieval, and so the type of indexing makes little difference: there is less than 10% variation among all the numbers. The main sources of this variation are probably the differences between the overheads of the various indexing methods.

To show the effect of queries that specify values only for some of the attributes in the key, we tested two simple queries on the daily schedule relation. The first specified the origin and the date, the second the destination and the date. The results appear in Table 3.

Table 3 shows the limitations of B-tree indexing. If the query does not specify a key attribute, B-tree indexing cannot make use of any attributes following the unspecified one in the key list. In this case, the key list on the schedule relation was "origin, destination, date." The first query achieves moderate performance because it uses only the first key attribute; the second query results in very bad performance because it cannot use any of the key attributes (this performance is worse than unindexed due to overheads). On the other hand, the multi-attribute hashing scheme on which the dsimc method is built is not at all sensitive to the order of keys, so it can exploit two key attributes for both queries. The difference between the performance of the two dsimc queries can be explained by the different number of answers they have in our database (11 for the From Melbourne query and 7 for the To Melbourne query); the two queries take about the same time per answer.

The daily schedule data occupy about 4.5 Mb of data in ASCII format. After the data are inserted into a Btree-indexed relation, they occupy about 7.4 Mb with about another 4.9 Mb required for the index. Similarly, the weekly schedule data occupy 95 Kb in ASCII; in Aditi it takes 164 Kb for the data file and 106 Kb for the index. We used the daily schedule relation, which contains 54,058 tuples, as test data in measuring the rate at which tuples can be inserted into relations. The insertion rates into relations without indexing, with B-tree indexing and with dynamic superimposed codeword indexing were roughly 790 tuples per second, 420 tuples per second, and 60 tuples per second, respectively. The lower speed of dsimc insertion is due to repeated copying of some tuples as the underlying linear hashed file doubles in size several times. This overhead can be avoided if the code knows in advance the number and average size of the tuples to be inserted.

Based on our experience with other tests, we can state that the effectiveness of many optimizations is strongly influenced by the number of iterations required to complete the bottom-up computation. Sometimes, a technique will introduce a one-off overhead that cannot be compensated for by the reduced cost per iteration because there are not enough iterations in the bottom-up computation. Presorting of relations is one such technique, although it can double performance on some queries. At other times, an optimization technique may speed up each iteration but require more iterations to answer a query. Unfortunately, one cannot always decide in advance whether a given optimization is worthwhile; the answer is often dependent on the query and the data. More experimentation is required to determine whether any useful heuristics exist. We envision the compiler exploiting such heuristics by generating code that switches at run-time, depending on the characteristics of the data, between two or more ways of evaluating a query.

6. Comparison with Other Systems

We now compare Aditi with four other deductive database systems, CORAL, LDL, Glue-Nail and EKS. All these systems have a syntax based on Prolog, and all support typical database operations such as aggregates.

CORAL (Ramakrishnan et al., 1992) is from the University of Wisconsin, Madison. It is a bottom-up system written in C++. While CORAL applications are intended to be written almost entirely in logic, application programmers can write code in C++ to create new abstract data types, and to define and manipulate CORAL relations. CORAL base relations may be stored in CORAL's own data file format or in the EXODUS (Carey et al., 1986) storage manager. Although CORAL is a single-user system, EXODUS is not; EXODUS views individual CORAL queries as transactions. (Updates generally have to be programmed in C++.) CORAL supports sets, and is unique in supporting multisets as well. It is also unique in supporting variables in facts and in allowing variables to occur in arbitrary positions in the tuples it manipulates. Since these abilities are important CORAL objectives, and since they make the implementation of relational algebra operations difficult, CORAL uses a specialized evaluation algorithm based more on Prolog technology than database technology. Although this algorithm contains many optimizations such as pipelining tuples from one operation to another, it also has some bottlenecks such as the frequent need for subsumption testing. On the other hand, the algorithm allows the evaluation of modularly stratified programs.

Glue-Nail (Phipps et al., 1991) is originally from Stanford. It is based on the earlier NAIL! system (Morris et al., 1986). Nail is a conventional logic programming language, extended with higher order syntax based on HiLog (the semantics is still first order). Its evaluation algorithm supports well-founded negation including fully general negation in the absence of function symbols. Glue is a new imperative language with primitives that operate on relations, and it is intended to be used directly by application programmers to write efficient database algorithms and to perform updates and I/O. (Other deductive database systems use existing languages such as C++ and Prolog to perform tasks that require side-effects, and their designers aim to automatically compile problem descriptions into code executing efficient algorithms.) Glue is similar to but at a higher level than RL, and since it is intended for human use it has a much nicer syntax. Glue-Nail is a self-contained and purely memory-based system; it performs disk I/O only to load relations into memory at startup and to save any modified relations at shutdown. It is a single user system, unlike the original NAIL! system, which compiled logic queries to SQL and used a commercial RDBMS to answer the SQL queries.

The LDL system (Chimenti et al., 1987; Naqvi and Tsur, 1989) is from MCC in Austin, Texas. It is also a single-user memory-based system that uses disk only for persistent storage between user sessions. Since LDL has no inherent link to any host language, updates are specified in the "declarative" rules. The time at which such updates take place (and therefore which parts of the whole computation see the effects of the update) is fixed by imposing an evaluation order. LDL was the first deductive database system to have built-in support for sets and set operations. LDL is currently being extended into LDL++, with a C++ interface to allow application programmers to define new abstract data types (as in CORAL) and an SQL interface to access and update existing RDBMSs (as in NAIL!).

EKS (Vieille et al., 1990) is from ECRC in Munich; it is derived from the earlier DedGin* (Vieille, 1988) system. Unlike the other systems, EKS uses top-down evaluation. EKS is the first deductive database system to offer direct support for hypothetical reasoning (evaluating a query in a temporarily modified database) and the related idea of post-conditional updates (modifying the database, testing a condition in the new database, and undoing the modification if the query fails). It is also currently unique in materializing derived relations (i.e., storing their extensions for faster retrieval). The incremental algorithm that updates materialized relations is also used for checking integrity constraints. EKS runs on top of the MegaLog persistent Prolog system (Bocca, 1991) that in turn is based on the BANG filesystem (Freeston, 1988). Even though MegaLog can store complex terms, EKS does not allow its applications to contain function symbols; in this it lags behind all of the other systems. On the other hand, MegaLog provides EKS with disk-based storage even for temporary relations and multi-user support with transactions.

Aditi is the only other system that can store (parts of) large temporary relations on disk and that supports multiple users. At the moment Aditi protects the integrity of its data with coarse-grain locks; a more fine-grained transaction system is currently being implemented. It supports complex terms containing function symbols but not variables. Aditi is unique among the five systems in being able to exploit parallelism. It is also unique in its ability to mix bottom-up set-at-a-time evaluation with top-down tuple-at-a-time computation.

7. Conclusion

Aditi is a practical deductive database system. The lower layers are based on relational technology wherever possible but with necessary modifications such as support for tuples not in first normal form. The upper layers are derived mostly from research specific to deductive databases. Many aspects in both layers build on original research by members of the Aditi team (Balbin and Ramamohanarao, 1987; Ramamohanarao and Shepherd, 1986; Ramamohanarao et al., 1988; Kemp et al., 1989, 1990, 1991, 1992; Balbin et al., 1991; Harland and Ramamohanarao, 1993; Kemp and Stuckey, 1993). Much of this research in turn used Aditi as a testbed in which to try out and evaluate ideas.

Aditi is a true database system. It is disk based, it is multiuser, and it can exploit parallelism in the underlying machine. It has both textual and graphical user interfaces, including an interface that understands SQL. It also includes interfaces to the NU-Prolog language, to which Aditi-Prolog is closely related. NU-Prolog code can call Aditi-Prolog; this link allows Aditi applications to create their own user interface. Aditi-Prolog code also can call NU-Prolog to compute some predicates top-down and tuple-at-a-time when such evaluation speeds up query processing.

Our projects for the future include:

- Integrating our prototype compiler, which generates code for non-stratified negation into the main Aditi system.
- Implementing high-level optimizations such as constraint propagation (Kemp et al., 1989) and the new transformation for linear predicates (Harland and Ramamohanarao, 1993).
- Implementing low-level optimizations such as redundant code elimination.
- Extending Aditi-Prolog by providing types and a sophisticated system of modes based on these types.
- Using the mode system to allow the handling of partially-instantiated data structures. We already know how to do this without subsumption tests (Somogyi et al., 1994).
- Adding object-oriented and higher-order features to Aditi-Prolog; we already have a draft language proposal.
- Finishing the implementation of the transaction system.
- Benchmarking Aditi against RDBMSs such as Oracle and Ingres on both relational and deductive type queries.

- Extending Aditi's parallel features to support distributed memory as well as shared-memory machines.

Those who would like to learn more about Aditi should refer to the Aditi users's guide (Harland et al., 1992*a*), the Aditi-Prolog language manual (Harland et al., 1992*b*), and the report describing the flights database from which the second set of tests (Section 5) was drawn (Harland and Ramamohanarao, 1992).

The Aditi system itself is available to interested researchers. It has already been distributed to about a dozen sites around the world.

Acknowledgments

Many people contributed significantly to Aditi, both by performing research and by developing software. We would like to acknowledge the contributions of Kim Marriott, John Shepherd, David Keegel, Warwick Harvey, Philip Dart, and Jeff Schultz in this respect. We thank Gill Dobbie for her comments on a draft of this article.

This research was supported by grants from the Australian Research Council through the Machine Intelligence Project, from the Victorian Department of Manufacturing and Industry Development, through the Collaborative Information Technology Research Institute, from the Australian Department of Industry, Technology and Commerce through the Key Centre for Knowledge-based Systems, and from the Australian Cooperative Research Center program through the Center for Intelligent Decision Systems.

References

Balbin, I., Port, G., Ramamohanarao, K., and Meenakshi, K. Efficient bottom-up computation of queries on stratified databases. *Journal of Logic Programming,* 11(3/4):295–345, 1991.

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

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

Beeri, C. and Ramakrishnan, R. On the power of magic. *Proceedings of the Sixth ACM Symposium on Principles of Database Systems,* San Diego, CA, 1987.

Bocca, J. Megalog: A platform for developing knowledge base management systems. *Proceedings of the Second International Symposium on Database Systems for Advanced Applications,* Tokyo, 1991.

Carey, M., DeWitt, D., Richardson, J., and Shekita, E. Object and file management in the EXODUS extensible database system. *Proceedings of the Twelfth International Conference on Very Large Databases,* Kyoto, Japan, 1986.

Chimenti, D., O'Hare, T., Krishnamurthy, R., Naqvi, S., Tsur, S., West, C., and Zaniolo, C. An overview of the LDL system. *IEEE Data Engineering*, 10(4):52–62, 1987.

Freeston, M. Grid files for efficient Prolog clause access. In: Gray, P. and Lucas, R., eds., *Prolog and Databases*, Chicester, England: Ellis Horwood, 1988, pp. 188–211.

Harland, J., Kemp, D.B., Leask, T.S., Ramamohanarao, K., Shepherd, J.A., Somogyi, Z., Stuckey, P.J., and Vaghani, J. Aditi-Prolog language manual. Technical Report 92/27, Department of Computer Science, University of Melbourne, Australia, November 1992*a*.

Harland, J., Kemp, D.B., Leask, T.S., Ramamohanarao, K., Shepherd, J.A., Somogyi, Z., Stuckey, P.J., and Vaghani, J. Aditi user's guide. Technical Report 92/26, Department of Computer Science, University of Melbourne, Australia, November 1992*b*.

Harland, J. and Ramamohanarao, K. Experiences with a flights database. Technical Report 92/28, Department of Computer Science, University of Melbourne, Australia, 1992.

Harland, J. and Ramamohanarao, K. Constraint propagation for linear recursive rules. *Proceedings of the Tenth International Conference on Logic Programming*, Budapest, Hungary, 1993.

Kemp, D.B. and Stuckey, P. Analysis-based constraint query optimization. *Proceedings of the the Tenth International Conference on Logic Programming*, Budapest, Hungary, 1993.

Kemp, D.B., Ramamohanarao, K., Balbin, I., and Meenakshi, K. Propagating constraints in recursive deductive databases. *Proceedings of the First North American Conference on Logic Programming*, Cleveland, OH, 1989.

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

Kemp, D.B., Stuckey, P.J., and Srivastava, D. Magic sets and bottom-up evaluation of well-founded models. *Proceedings of the 1991 International Logic Programming Symposium*, San Diego, CA, 1991.

Kemp, D.B., Stuckey, P.J., and Srivastava, D. Query restricted bottom-up evaluation of normal logic programs. *Proceedings of the Joint International Conference and Symposium on Logic Programming*, Washington, DC, 1992.

Knuth, D.E. Sorting and searching. Vol. 3, Chapter 5.4.1, *The Art of Computer Programming*, Reading, MA: Addison-Wesley, 1973.

Leask, T.S., Ramamohanarao, K., and Stuckey, P.J. Exploiting parallelism in bottom-up computation in Aditi. *Proceedings of the ILPS Workshop on Deductive Databases*, San Diego, CA, 1991.

Linderman, J. Theory and practice in the construction of a working sort routine. *Bell Laboratories Technical Journal*, 63(8(part 2)):1827–1843, 1984.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems,* 17(1):94–162, 1992.

Morris, K., Naughton, J.F., Saraiya, Y., Ullman, J.D., and Gelder, A.V. YAWN! (yet another window on NAIL). *IEEE Data Engineering,* 10(4):28–43, 1987.

Morris, K., Ullman, J.D., and Gelder, A.V. Design overview of the NAIL! system. *Proceedings of the Third International Conference on Logic Programming,* London, 1986.

Mumick, I.S., Finkelstein, S.J., Pirahesh, H., and Ramakrishnan, R. Magic conditions. *Proceedings of the Ninth Symposium on the Principles of Database Systems,* Nashville, TN, 1990.

Nakayama, M., Kitsuregawa, M., and Takagi, M. Hash-partitioned join method using dynamic destaging strategy. *Proceedings of the Fourteenth Conference on Very Large Databases,* Los Angeles, 1988.

Naqvi, S. and Tsur, S. *A Logical Language for Data and Knowledge Bases.* New York: Computer Science Press, 1989.

Ousterhout, J.K. *Tcl and the Tk Toolkit.* Reading, MA: Addison-Wesley Publishers, 1994.

Phipps, G., Derr, M., and Ross, K. Glue-Nail: A deductive database system. *Proceedings of the Tenth ACM Symposium on Principles of Database Systems,* Denver, CO, 1991.

Port, G., Balbin, I., and Ramamohanarao, K. A new approach to supplementary magic optimisation. *Proceedings of the First Far-East Workshop on Future Database Systems,* Melbourne, Australia, 1990.

Ramakrishnan, R., Srivastava, D., and Sudarshan, S. Rule ordering in bottom-up fixpoint evaluation of logic programs. *Proceedings of the Sixteenth International Conference on Very Large Databases,* Brisbane, Australia, 1990.

Ramakrishnan, R., Srivastava, D., and Sudarshan, S. CORAL: A deductive database programming language. *Proceedings of the Eighteenth International Conference on Very Large Databases,* Vancouver, Canada, 1992.

Ramamohanarao, K. and Shepherd, J. A superimposed codeword indexing scheme for very large Prolog databases. *Proceedings of the Third International Conference on Logic Programming,* London, 1986.

Ramamohanarao, K. and Shepherd, J. Partial match retrieval for dynamic files using superimposed codeword indexing. In: Balaguruswamy, E. and Sushila, B., eds., *Computer Systems and Applications: Recent Trends.* New Dehli, India: McGraw-Hill, 1990.

Ramamohanarao, K., Shepherd, J., Balbin, I., Port, G., Naish, L., Thom, J., Zobel, J., and Dart, P. The NU-Prolog deductive database system. In: Gray, P. and Lucas, R., eds. *Prolog and Databases,* Chicester, England: Ellis Horwood, 1988, pp. 212–250.

Sacca, D. and Zaniolo, C. The generalized counting method for recursive logic queries. *Proceedings of the International Conference on Database Theory,* Rome, 1986.

Sacca, D. and Zaniolo, C. Implementation of recursive queries for a data language based on pure horn logic. *Proceedings of the Fourth International Conference on Logic Programming,* Melbourne, Australia, 1987.

Somogyi, Z., Kemp, D.B., Harland, J., and Ramamohanarao, K. Subsumption-free bottom-up evaluation of logic programs with partially instantiated data structures. *Proceedings of the Fourth International Conference on Extending Database Technology,* Cambridge, England, 1994.

Tham, J. Duplicate removal and parallel join algorithms. Technical Report/Honours Report, Department of Computer Science, University of Melbourne, Australia, 1993.

Thom, J.A. and Zobel, J.A. NU-Prolog reference manual, Version 1.3. Technical Report, Department of Computer Science, University of Melbourne, Australia, 1988.

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

Vaghani, J., Ramamohanarao, K., Kemp, D.B., Somogyi, Z., and Stuckey, P.J. Design overview of the Aditi deductive database system. *Proceedings of the Seventh International Conference on Data Engineering,* Kobe, Japan, 1991.

Vieille, L. Recursive query processing: Fundamental algorithms and the DedGin system. In: Gray, P. and Lucas, R., eds., *Prolog and Databases.* Chicester, England: Ellis Horwood, 1988.

Vieille, L., Bayer, P., Küchenhoff, V., and Lefebvre, A. EKS-V1: A short overview. *Proceedings of the AAAI Workshop on Knowledge Base Systems,* Boston, MA, 1990.