Proceedings of the Workshop on Programming with Logic Databases

Raghu Ramakrishnan

Technical Report #1183

October 1993

Proceedings of the Workshop on Programming with Logic Databases In Conjunction with ILPS Vancouver, B.C., October 30, 1993

Editor: R. Ramakrishnan
Computer Sciences Department
University of Wisconsin-Madison, WI 53706, U.S.A.

Abstract

An increasing number of deductive systems are now either available or becoming available (e.g., Aditi, CORAL, EKS, LDL, LOGRES, LOLA, NAIL-Glue, RDL, XSB). The workshop schedule features twelve papers describing a range of applications, and demo presentations of several implemented systems. The proceedings contain all presented papers, and short demo descriptions. The papers are in order of presentation, followed by demo descriptions; included are descriptions of two demos that are not on the presentation schedule.

r 183

Workshop on Programming with Logic Databases

Vancouver, B.C., Canada, October 30, 1993
In Conjunction with ILPS 93

An increasing number of deductive systems are now either available or becoming available (e.g., Aditi, CORAL, EKS, LDL, LOGRES, LOLA, NAIL-Glue, RDL, XSB). A number of applications have been developed using these systems, typically by the system developers, although this should change as the systems are more widely distributed. It is clear that writing effective programs requires users to understand some broad principles, and differs in significant ways from writing programs in languages like C, or even Prolog. In systems that also provide some support for object-orientation, developing good programming techniques is even more important. Further, the scope of the domains for which these systems provide a good platform is little understood. Understanding these issues better, and developing substantial applications in a variety of domains, is essential to the future of the field.

The goal of this workshop is to provide a forum for users and implementors of deductive systems to share their experience. The emphasis will be on the use of deductive systems — all papers deal with actual programs in some direct way. The schedule features twelve papers describing a range of applications, and demo presentations of several implemented systems. (Additional demos will proceed in parallel with the scheduled presentations.) The proceedings contains all presented papers, and short demo descriptions. The papers are in order of presentation, followed by demo descriptions in order of presentation; there are also descriptions of two demos that are not on the presentation schedule, but will proceed in parallel with the presentation track.

This workshop was made possible through the efforts of a number of people. Eric Kolotyluk at SFU organized the hardwarde for the demonstrations, Bernie Kowey helped with registration and local arrangements (including an extension in the room availability!), Francesca Rossi co-ordinated the workshop program, and Jiawei Han did a fine job of local arrangements, in addition to being on the program committee. The program committee members reviewed submitted papers at very short notice and ensured that the workshop features a strong technical program.

I thank everyone involved for the work and time that they have generously contributed to the organization of this workshop.

Raghu Ramakrishnan
University of Wisconsin-Madison

Program Committee:

Oris Friesen (Bull)	Raghu Ramakrishnan (U. Wisconsin)
Jiawei Han (SFU)	S. Sudarshan (Bell Labs)
David Kemp (U. Melbourne)	Dick Tsur (SBOC and U. Texas)
Jerry Kiernan (IBM Almaden)	Laurent Vieille (Bull)
Werner Kiessling (U. Muenchen)	Carlo Zaniolo (UCLA)
Inderpal Mumick (Bell Labs)	

Organizer	**Local Arrangements**
Raghu Ramakrishnan (U. Wisconsin)	Jiawei Han (SFU)
raghu@cs.wisc.edu	han@cs.sfu.ca

Schedule of Presentations [1]

SESSION 1: 8:00 – 10:00

Overview of the Workshop: 8:00 – 8:20

Talks: 8:20 – 9:40

An Aditi Implementation of a Flights Database, *J. Harland and K. Ramamohanarao (Dept. of CS, Univ. of Melbourne)*

What One Genome-Mapping Lab Needs From Its Database, *N. Goodman, S. Rozen and L. Stein (Whitehead Institute for Biomedical Research, MIT)*

MIMSY: A System for Analyzing Time Series Data in the Stock Market Domain, *W.G. Roth, R. Ramakrishnan and P. Seshadri (CS Dept., Univ. of Wisconsin-Madison)*

Efficient Visual Queries for Deductive Databases, *D. Vista (Dept. of CS, Univ. of Toronto) and Peter T. Wood (Univ. of Cape Town)*

Demo Presentation 1: 9:40 – 10:00

ADITI, *J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask and J. Harland (Univ. of Melbourne)*

Coffee Break: 10:00 – 10:20

SESSION 2: 10:20 – 12:20

Demo Presentation 2: 10:20 – 10:40

CORAL, *R. Ramakrishnan and P. Seshadri (Univ. of Wisconsin-Madison), D. Srivastava and S. Sudarshan (AT&T Bell Labs)*

[1]The papers in the proceedings are ordered as in the schedule. Short demo descriptions are included at the end.

Talks: 10:40 – 12:00

Improving Data Quality Via LDL++, *S. Shaw, L. Foggiato-Bish, I. Garcia, G. Tillman, D. Tryon and W. Wood (Pacific Bell), C. Zaniolo (CS Dept., UCLA)*

Using LDL++ for Spatio-temporal Reasoning in Atmospheric Science Databases, *R.R. Muntz, E.C. Shek and C. Zaniolo, (CS Dept., UCLA)*

Developing Applications With CORAL, *A. Saran, K. Park, Y. Chen, A.P. Aguiar, T.R. Smith, J. Su (Dept. of CS, UCSB)*

A Declarative Language Environment for Knowledge-Workers, *O. Friesen (Groupe Bull, Phoenix)*

Demo Presentation 3: 12:00 – 12:20

GLUE/NAIL, *M. Derr (AT&T Bell Labs) and Geoff Phipps (Sun Microsystems)*

Lunch: 12:20 – 1:30

SESSION 3: 1:30 – 3:30

Demo Presentation 4: 1:30 – 1:50

XSB, *K. Sagonas, T. Swift and D.S. Warren (Dept. of CS, SUNY at StonyBrook)*

Programming the PTQ Grammar in XSB, *D.S. Warren (Dept. of CS, SUNY at StonyBrook)*

AMOS: A Natural Language Parser in LOLA, *G. Specht, B. Freitag and H. Schuetz (Institut fur Informatik, Technische Universitat Muenchen)*

A New User's Impressions on LDL++ and CORAL, *P. Hsu and C. Zaniolo (CS Dept., UCLA)*

LogicBase: A Testbed for Deductive Query Evaluation, *J. Han, L. Liu and Z. Xie (School of Computing Science, Simon Fraser University)*

Demo Presentation 5: 3:10 – 3:30

LOGICBASE, *J. Han, L. Liu, Z. Xie (School of Computing Science, Simon Fraser University)*

Break: 3:30 – 3:50

Demo Presentation 6: 3:50 – 4:10

LOLA, *B. Freitag, H. Schuetz and G. Specht (Institut fur Informatik, Technische Universitat Muenchen)*

Panel Discussion: 4:10 – 5:30

An Aditi Implementation of a Flights Database

James Harland and Kotagiri Ramamohanarao
Department of Computer Science
University of Melbourne
{jah,rao}@cs.mu.oz.au

1 Introduction

A common example application for a deductive database system is a flights database, i.e. a system that can answer questions such as "What is the cheapest flight between Melbourne and New York?". This appears to be a relatively simple problem, as it is merely finding the shortest distance between two nodes in a weighted directed graph. However, in practice there are several implicit constraints and default preferences which complicate the picture; generally, a passenger will usually want a flight which is as cheap as possible, minimises the time spent in transit, leaves on a specified date and arrives before a certain deadline. There also may be circumstances in which the passenger is prepared to accept a seemingly sub-optimal flight in order to satisfy a particular preference. For example, John may ask a travel agent to book him the cheapest flight from Melbourne to London. When told that this will involve a twelve hour wait in transit in Singapore, he then asks that he travel via Bangkok, even though it will cost an extra \$100. However, when told that only Shonky Airlines flies to Bangkok, he decides to spend an extra \$300 to travel with Deluxe Airways, even though there are two cheaper flights.

In this paper we describe our experiences with an implementation of such an application in the Aditi deductive database system, which has been developed at the University of Melbourne [2] (see the references in [2] for more 'on deductive databases per se). This has become the standard demonstration for the Aditi system. There are several reasons why a flights database seems to be a good choice for such a demonstration.

Firstly, recursive rules are needed. In our experience, it seems that deductive databases will not become commercially accepted until they are perceived to be not only at least as reliable, robust and efficient as relational systems, but also to represent a significant advance in technology. As a result, an example implementation, such as the Aditi flights database, will need to incorporate features that relational systems do not have, and recursive rules appears to be the best example of such a feature.

Secondly, the recursive rules involved are not those for transitive closure. Such rules can be mimicked by transitive closure operators in languages such as SQL, and often appear to be toy-like. Clearly the transitive closure of the basic flights relation is not only massive, it is generally fairly useless. In addition, the problem of finding a flight schedule is not simply a path minimisation problem either. For example, a passenger may not want the cheapest or shortest flight overall, but the cheapest flight which avoids certain airlines, or minimises the amount of time spent at intermediate stops.

Thirdly, large amounts of information are involved. For example, the worldwide timetable for British Airways alone fills a book of around 250 A5 pages. This means that there is no shortage of data for testing purposes or for demonstrating the abilities of the system.

Fourthly, there is a variety of constraints associated with queries. For example, a passenger may wish to leave no earlier than 8.00am, unless he can save more than $100 by doing so. In a deductive database system, such constraints can be expressed simply and easily.

Finally, the problem is a natural one and easily understood. For these reasons we believe that a flights database is an appropriate demonstration of the abilities of a deductive database system.

This paper is organized as follows. In Section 2 we give a brief introduction to the Aditi system and the Nu-Prolog interface to Aditi. In Section 3 we describe the data design for the flights database, and in the following section the description of the rules of the system. In Section 5 we present some performance results for the system, and finally in Section 6 we present our conclusions.

2 Aditi

2.1 The Aditi System

Aditi[1] is a deductive database system which has been developed at the University of Melbourne. Programs in Aditi consist of base relations (facts) together with derived relations (rules), and are in fact a subset of (pure) Prolog. Queries are a conjunction of atoms (as in Prolog), and a bottom-up evaluation technique is used to answer queries. In finding all answers to a given query, Aditi, like many deductive database systems, uses algorithms and techniques developed for the efficient answering of queries in relational database systems. Thus we expect that Aditi need not be less efficient than a relational system for purely relational queries. Aditi also uses several optimisation techniques which are peculiar to deductive databases, particularly for the evaluation of recursive rules. These techniques include magic sets, supplementary magic sets, semi-naive evaluation, predicate semi-naive evaluation, the magic sets interpreter and the context transformation [1].

[1]Aditi is named after the goddess in Indian mythology who is "the personification of the infinite" and "mother of the gods".

Aditi is based on a client/server architecture, in which the user interacts with a front-end process, which then communicates with a back-end server process which performs the database operations. There are three kinds of server process in Aditi: the query server, which manages the load that Aditi places on the host machine, database access processes, one per client, which control the evaluation of the client's queries, and relational algebra processes, which carry out relational algebra operations such as joins, selections and projections on behalf of the database access processes.

There are four main characteristics of Aditi which, collectively, distinguish it from other deductive databases: it is disk-based, which allows relations to exceed the size of main memory; it supports concurrent access by multiple users; it exploits parallelism at several levels; and it allows the storage of terms containing function symbols. It has been possible for researchers to obtain a beta-test version of Aditi since January 1993, and a full release of the system is expected soon. The current version of Aditi comes with a Prolog-like (text-based) interface, a graphical user interface, interfaces to both SQL and Ingres and a programming interface to Nu-Prolog. It is also possible to embed top-down computations within Aditi code.

2.2 The Nu-Prolog Interface

A useful feature of Aditi is that there is a "two-way" interface between Aditi and Nu-Prolog, in that a Nu-Prolog program can make call to Aditi, and an Aditi program can make calls to Nu-Prolog. In this way a Prolog program can be used either as a pre- (or post-) processor for Aditi, or as a tool for intermediate computation within Aditi. This interface is transparent, in that a call to Aditi within a Nu-Prolog program appears just like any other Nu-Prolog call, and a call to Nu-Prolog within an Aditi program looks just like any other Aditi call. This makes it very simple to transfer code between Aditi and Nu-Prolog and vice-versa(provided, of course, that there are no termination problems introduced by the switching of execution mechanism). For example, to find a list of connections using the Aditi relation `paths` and then using Nu-Prolog to reverse the list, one would use the code below.

```
paths(X,Y,Paths), reverse(Paths, RevPaths)
```

where `reverse` is the usual Nu-Prolog reverse predicate. This code would remain the same if the `reverse` predicate was written in Aditi and the `paths` predicate was written in Nu-Prolog.

Sometimes a programmer may desire more control than is possible in the transparent interface. For example, it may be useful to make a call to Aditi to determine what flights satisfy a given constraint, and then sort and pretty-print all these flights. For purposes such as this, it is also possible to access Aditi from Nu-Prolog via a table and cursor mechanism. When a call to Aditi is made via the `dbQuery` predicate, a handle is returned, which can be used to obtain a cursor, i.e. a pointer to the next tuple in the answer relation. This cursor can then be used to step through the relation as many times as required. This not only gives the Nu-Prolog programmer more control over the answers to Aditi queries, but may also improve performance for certain applications.

The embedding of Aditi within Nu-Prolog means that we can have a Nu-Prolog program prompt for input, pose the query to the Aditi system, sort the answers according to some user-specified preference, and display the flight information for each answer in an interactive and meaningful way. This not only provides more flexibility, but also makes it possible to connect existing systems to Aditi in a simple way. The ability to call Nu-Prolog from Aditi means that we can mix top-down and bottom-up computations. For example, in the flights database there is a need to calculate the day of the week corresponding to a given date. The program to compute the day from the date is not particularly large, but contains a significant number of tests (such as whether the given year is a leap year or not) and very little data. Whilst this program may be evaluated in either a bottom-up or top-down fashion, it turns out that top-down evaluation is significantly more efficient, and hence we make a call to this top-down program from the bottom-up one when the conversion is required. In our experience list processing predicates are also generally significantly more efficient when evaluated top-down than bottom-up. In this way we can make good use of existing efficient code within the database engine.

3 Flight Information

There are several pieces of information associated with a given flight — its origin and destination, the date and time of departure, the flight number, and so forth. Given that a passenger may wish to travel between two arbitrary places, when it comes to storing this information, it seems natural to store this information for each "hop"; in other words, whilst a given plane may travel from Melbourne to Sydney to Auckland to Honolulu to Los Angeles to Chicago, a passenger may wish to travel from Auckland to Honolulu. Hence we would store this particular sequence as five separate flights, rather than one long flight. This also simplifies the problem of pricing information, as otherwise we would have to know how to divide up a given price into sub-component. This, of course, is a trade-off, in that in order to make answering queries faster we increase the storage needed. Given the large amounts of information involved we cannot be too blasé about storage requirements, but this particular trade-off seems to be worthwhile. This method of storage also seems to be an appropriate level of granularity, in that a passenger wishing to go to a somewhat obscure place may need to travel to an intermediate hub, and from there to the destination. Storing each "hop" means that such the requisite search can be performed in a more straightforward manner than if each complete plane journey was used as the base unit of information.

Another consideration is the patterns involved in airline schedules. Whilst timetables change from time to time, and vary with expected demand from season to season, many schedules are organised on a weekly basis, in that for a specified period of time, the schedule is the same from week to week. This means that rather than store each flight individually, which would involve large amounts of repetition, we store the weekly schedule once, and then determine whether there is a flight on a given date from the appropriate weekly schedule. This, again, is a trade-off, in that it involves significantly less storage, but makes searching for a flight more involved. However, as the extra computation is to determine the day of the week corresponding to a given date, this again seems like a worthwhile compromise, as it would seem that performing this calculation is preferable to storing each flight individually. For these reasons, flight information is stored in a relation `flight_weekly` which has ten attributes, given below:

Origin, Destination, Dtime, Atime, Incr, Airline, Number, Day, V_from, V_to

where the flight is from Origin to Destination, departing at Dtime and arriving at Atime. Incr indicates how many days later than departure the flight arrives; this will often be 0, but can also be 1, and sometimes 2. Airline and Number specify the airline and flight number respectively, and Day the day of the week that the flight leaves. V_from and V_to give the dates between which the schedule is valid. Thus a Qantas flight from Sydney to Los Angeles which departs four times a week during the (southern) winter would be represented as the following four tuples:

```
sydney,los_angeles,2000,1900,0,qantas,qf007,wed,date(1,6,1993),date(31,8,1993)
sydney,los_angeles,2000,1900,0,qantas,qf007,fri,date(1,6,1993),date(31,8,1993)
sydney,los_angeles,2000,1900,0,qantas,qf007,sat,date(1,6,1993),date(31,8,1993)
sydney,los_angeles,2000,1900,0,qantas,qf007,sun,date(1,6,1993),date(31,8,1993)
```

Thus to determine whether such a flight departs on the 11th of August 1993, all we need do is determine which day of the week this is. As it is a Wednesday, we find that there is indeed such a flight on that day, being Qantas flight 7, departing at 8pm, and arriving in Los Angeles at 7pm on the same day.

The way that such a query would be expressed is as follows:

```
?- day(date(11,8,1993), Day), flight(sydney, los_angeles, Dtime, Atime,
Incr, Airline, Number, Day, V_from, V_to), between(V_from, date(11,8,1993),
                                  V_to).
```

where day is a relation between dates and days of the week, and `between(D1, D2, D3)` is true if `D2` is a date no earlier than `D1` and no later than `D3`. In the system we incorporate such queries into the `flight_between` relation.

4 Assembling a Flight Schedule

Retrieving flight information is one thing; putting together a schedule is quite another — not only do we need to satisfy constraints such as allowing a minimum time between flights (as well as any constraints given by the passenger), but due to the large amounts of information involved, we also need to ensure that the search procedure is feasible. This is particularly acute when there is no direct flight from the origin to the destination, as the choice of intermediate stops is, in principle, vast, but in practice there are usually only a few realistic choices. For example, if a passenger wishes to travel from Melbourne to Los Angeles, we need only consider trans-Pacific flights, and there is no need to consider flights to Europe or Asia. Hence the search procedure should avoid consideration of flights via London, Bombay or Tokyo, but presumably consider flights via Sydney, Auckland or Honolulu. Various heuristics may be employed to determine the most suitable routes, such as the locating the nearest city with a direct flight to the destination, travelling to the nearest "hub" airport, or finding the route which is shortest in time and/or distance. Given that such heuristics will vary according to local airline policies and may change with time, we have chosen to use a single relation to guide the search. This relation specifies which intermediate stops are feasible for a given origin and destination. The intention of the `feasible` relation is to exclude from consideration routes which are clearly of no benefit. In the above example, the `feasible` relation includes the tuples

```
melbourne, auckland, los_angeles
melbourne, sydney, los_angeles
melbourne, honolulu, los_angeles
```

but London, Bombay and Tokyo are not included as feasible intermediate stops for a trip between Melbourne and Los Angeles.

This relation is defined using both a base relation and a set of rules. This allows us to take advantage of some regularities in the feasible routes, such as commutativity. For example, it is feasible to travel from Melbourne to Auckland to Honolulu to Los Angeles, and hence it is feasible to travel in the reverse direction as well. In addition, it is simple to state principles such as "Honolulu is a feasible stop for all flights from Melbourne to somewhere in North America" as rules. Such a rule is given below.

```
            feasible(melbourne, honolulu, Z) :- place(Z, north_america).
```

Thus the default code to find a flight schedule is as follows:

```
trip(From, To, Ddate, Earliest, Latest, Stime, Ddate1, Dtime, Adate, Atime,
        [flight(From, To, Ddate1, Dtime, Adate, Atime, A, N)]) :-

        flight_between(From, To, Ddate, Earliest, Latest,
                Ddate1, Dtime, Adate, Atime, A, N).

trip(From, To, Ddate, Earliest, Latest, Stime, Ddate2, Dtime, Adate1, Atime1,
        flight(Stop,To,Ddate1,Dtime1,Adate1,Atime1,A,N).F) :-

        feasible(From, Stop, To),
        trip(From, Stop, Ddate, Earliest, Latest, Stime,
                Ddate2, Dtime, Adate, Atime, F),
        NewEarliest = Atime + 100,
        NewLatest = Atime + Stime,
        flight_between(Stop, To, Adate, NewEarliest, NewLatest,
                Ddate1, Dtime1, Adate1, Atime1, A, N).
```

Note that trip is similar to the transitive closure of the flight_between relation, but it is actually subtlely different, due to the restriction that each "hop" must be feasible. The first five arguments to flight_between must be bound, specifying the desired origin, destination, departure date, and the earliest and latest times of departure. The other arguments return the actual flight information, including departure and arrival times, the airline and flight number, and the number of "hops" used in the trip. Note also that the arrival date is given as well, as for international flights, this may be different from the departure date. In fact, the departure date is given as well, which is for reasons of extensibility. Currently the system only searches within a departure "window" of 48 hours or less; longer periods will not result in any more flights being found. It is possible to extend this to be any interval, so that searches such as that for a flight any time in the next four days might be simply expressed.

The first six arguments to trip must be bound, not only to include the information passed to flight_between, but also to include the maximum amount of time the customer is prepared to wait in transit. Once the recursive call to trip has returned, the window is then shifted, so that there is at least one hour between flights.

Another way in which heuristics can be useful is to use the same airline for the duration of the trip. This will often reduce the cost, as well as simplify travel arrangements. Also, the passenger may wish to specify a particular airline, due to a personal preference or some inducement such as a frequent flyer discount.

The way that this is expressed in the system is to add an extra argument to the `trip` relation, so that the recursive rule becomes

```
trip_air(From, To, Ddate, Earliest, Latest, Stime, Airline,
        Ddate2, Dtime, Adate1, Atime1,
        flight(Stop,To,Ddate1,Dtime1,Adate1,Atime1,Airline,N).F) :-

        feasible(From, Stop, To),
        trip_air(From, Stop, Ddate, Earliest, Latest, Stime, Airline,
                Ddate2, Dtime, Adate, Atime, F),
        NewEarliest = Atime + 100,
        NewLatest = Atime + Stime,
        flight_between(Stop, To, Adate, NewEarliest, NewLatest,
                Ddate1, Dtime1, Adate1, Atime1, Airline, N).
```

Note that the variable `Airline` in both the call to `trip_air` and `flight_between` ensures that the airline is the same for each "hop".

Calls to `trip_air` are expected to have the first six variables bound, specifying the origin, destination, and the desired departure date, as well as the earliest and latest times on that day that the passenger is prepared to fly, and the maximum time to be spent at each intermediate stop. The seventh argument may be either given, or will be computed — the former case when the passenger has a particular airline in mind and nominates it, and the latter case when the passenger has no particular choice, provided that the same airline is used all the way. Thus a typical query to this rule would be:

```
trip_air(melbourne, los_angeles, date(1,11,1993), 1000, 2200, 200, qantas,
         Ddate, Dtime, Adate, Atime, Flights)
```

Thus all tuples returned for this query will be those representing Qantas flights. Note that all these tuples will be returned, amongst others, as answers for the query

```
trip_air(melbourne, los_angeles, date(1,11,1993), 1000, 2200, 200, Airline,
         Dbate, Dtime, Adate, Atime, Flights)
```

This is a good example of the use of logic programming techniques for this application, in that the airline argument may be either instantiated, in which case the binding is used to narrow the search, or left uninstantiated, in which case values are found for the variable.

5 Performance Results

In this section we present some performance measurements on the flights database. The four queries we are reporting results on all involve a hypothetical traveller called Phineas Fogg who wants to travel around the world as fast as possible. 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.

We have two implementations of the code that finds trips (sequences of flights) between cities. 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. The reason why we did not include data for the case when the trip-finding predicate is compiled without optimization is that that predicate is allowed only with respect to queries that specify the starting-date argument, and therefore the predicate cannot be evaluated bottom-up without first being transformed by a magic-like optimization. The test results appear in table 1, whose speedups are computed with respect to the magic transformed program using no indexing. Speedups for a given query follow the time and the colon.

The table 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

Results for Phineas Fogg 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 Phineas Fogg 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

Table 1: Results for Phineas Fogg queries

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 due to two reasons. First, the tours with more answers are those that visit fewer regions and thus call `trip` a smaller number of times. Second, the cost of the joins invoked by `trip` depend 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-fold to about 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. 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.

6 Conclusion

It is our belief that deductive database systems will need to be demonstrably impressive in order to gain commercial acceptance, and that a flights database, like the Aditi version of one described in this paper, is a good example of the application of techniques peculiar to deductive database systems. We have demonstrated this system to people interested in finding out more about deductive database systems, and the reaction has been generally quite positive. The system has also been a useful test for Aditi. For example, we have used a flight relation containing 54,000 tuples to test the way that Aditi reacts under load. Whilst this particular application has been produced by the developers of Aditi, it seems that such a demonstration system is necessary in order to get others interested in using deductive database systems in a significant way. We are actively encouraging such interaction, as well as pursuing further Aditi applications.

References

[1] D. Kemp, K. Ramamohanarao and Z. Somogyi, Right-, Left- and Multi-linear Rule Transformations which Maintain Context Information, *Proceedings of the Sixteenth*

International Conference on Very Large Databases 380-391, Brisbane, August, 1990.

[2] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, P. Stuckey, T. Leask, J. Harland, The Aditi Deductive Database System, Technical Report 93/10, Department of Computer Science, University of Melbourne, 1993.

Requirements for a Deductive Query Language in the MapBase Genome-Mapping Database [1]

Nat Goodman, Steve Rozen, Lincoln Stein

{nat,steve,lstein}@genome.wi.mit.edu
Whitehead Institute for Biomedical Research
One Kendall Square
Cambridge MA 02139

Abstract

MapBase is a database that stores information about short DNA sequences called markers and about the positions of markers along the chromosomes of an organism. It also stores information about experimental steps carried out on the markers. For modeling and performance reasons, we built MapBase on top of a C++-based object-oriented database management system. We describe MapBase's current design, and examine the requirements that, together with the underlying database management system, have shaped it. Analysis of the design's strengths and weaknesses indicates that MapBase requires improved support from the underlying database management system. We conclude that MapBase's most pressing needs are for a more powerful ad hoc query facility and for more flexibility in schema evolution. We believe that a deductive query language could provide the support we need and greatly improve the integration between MapBase and its clients.

1 Overview and Requirements

MapBase is a critical component of the genome-mapping efforts at the Whitehead Institute/MIT Center for Genome Research. These efforts will require completion of over 2.5 million experiments, each of which requires several steps. Broadly speaking, the purpose of these experiments is to find short DNA sequences called "markers" and to determine their locations on the chromosomes of an organism, thereby generating a genome map. MapBase records both the experimental steps and the conclusions about markers derived from these experiments. For more details on MapBase's role in genome mapping please see [Goo93, GRS92].

Several critical requirements have driven MapBase's current design:

R-1. The need to model complex data types, for example DNA sequences and genome maps. DNA sequences are like strings, but they have a restricted

[1]This work was supported by funds from the National Institutes of Health, National Center for Human Genome Research, grant number P50 HG00098.

alphabet, and require special operations, such as reverse complementation (see below), that can't easily be provided in today's relational systems. A genome map is, abstractly, a sequence of groups of markers with inter-group distances. The map must also record multiple possible positions of markers for which experimental data is inconclusive.

R-2. The need to accommodate rapid changes in experimental protocols as genome-mapping technology evolves. In essence, the Genome Center is engaged in continual process re-engineering. As a result, MapBase requires a major schema change at least every two months, and the frequency of schema changes is, if anything, increasing. In this respect, MapBase's requirements differ from those of some other (deductive) molecular biology databases (e.g. [SGT, GST93]), which are primarily concerned with querying experimental results.

R-3. The need for a client/server architecture with the ability to supply data to clients written in many different languages. In addition to the MapBase database, the system includes a map construction program called MAPMAKER[LGA+87], a program called PRIMER [LDL91] for analyzing markers, numerous small application programs, user interface programs, and programs to control laboratory machines. For pragmatic and historical reasons, these programs are written in a variety of programming languages: C, C++, Smalltalk, Perl, and Lisp. The Macintosh is the desktop platform of choice among our users, who often enter MapBase data by means of Excel spreadsheets created on these machines. Most data analysis programs run on Unix workstations, although a few programs run only on Macintoshes or PCs.

R-4. The need for high performance. Our users' response-time expectations are high; they are accustomed to having computers on their desks, and they expect to be able to access MapBase as quickly as they can access local data on their desktop Macintosh. Our peak retrieval load requires the transfer of 140,000 bytes to a particular application program; the system accomplishes this in about 30 seconds. We expect the peak update and retrieval rates to grow by a factor of 10 over the life of the project.

R-5. The need for concurrent multi-user access. Different users must be able retrieve data from MapBase and update it concurrently.

R-6. The need for reliability. Mapbase must be able to survive both soft failures and disk failures without loss of data. It is also necessary to ensure that the data in MapBase make sense. This will be particularly important as the number of different client applications updating MapBase grows.

In addition to the requirements above, we have one meta-requirement. We wish to avoid becoming locked into any one OODBMS. OODBMSs are new technology, and we want to be able to take advantage of future improvements regardless of the system that offers them.

It's worth mentioning at this point why we think deductive query languages could be important for MapBase. As we discuss below, our current design falls short of fully satisfying our requirements for schema flexibility (R-2) and a client/server architecture (R-3). A deductive query language would let us pose many ad-hoc queries that we now would have to code imperatively and compile into the database. Although most MapBase queries are not recursive, some are, and it is possible that if we had a deductive query language we would pose more recursive queries. Furthermore, if the deductive query language were extensible by providing support for user-defined functions and abstract data types, we would have the option of prototyping queries in the deductive language, and then, if performance requirements so dictate, move performance-critical predicates into the database.

In the following sections we discuss how MapBase's design tries to satisfy these needs using a C++-based object-oriented database management system (OODBMS). We analyze the strengths and weaknesses of the current design, and describe additional support that MapBase will require from its underlying database management system.

2 Current Design

We decided to build MapBase on top of an OODBMS because of our requirements for data modeling (R-1) and performance (R-4). We also considered a relational database management system, Sybase [MD92], which is the most widely used relational system in molecular biology research. We decided, however, that, in the long term, it would be too hard to satisfy our modeling requirement, R-1, with Sybase. We chose ObjectStore because we judged it to be the most mature of the OODBMSs at the time the choice was made. We also considered ONTOS[ONT92], O_2[$O_2$91], VERSANT[Ver93], and GemStone[BOS91].

Because ObjectStore offers the ability to store instances of any C++ class, we were able to model complex data types, as required by R-1. In addition, ObjectStore provides excellent single user performance, provided the database fits in a main memory cache, which helped satisfy our requirement R-4, for high performance.[2]

Having selected ObjectStore, we were faced with a number of further issues:

[2]Performance degrades rapidly when the database exceeds the cache. However, ObjectStore is designed to exploit very large caches and allows the system administrator to easily control the cache size. As the database grows, we deal with the performance impact by buying more memory and increasing the cache size.

I-1. Multi-user performance is troublesome because of lock contention. We found it difficult to avoid hot spots in storage allocation and other low-level areas. We experimented with ObjectStore's versioning capability, but in our tests this imposed too great a performance overhead on both queries and updates.

I-2. ObjectStore offers no roll-forward recovery. Therefore a disk crash or a dirty crash of the database management system can destroy all updates since the last database backup.

I-3. In the first release of ObjectStore that we used there was no system support for schema evolution.[3] Although the use of C++ classes made extending the schema conceptually simple, a schema change could invalidate the database and make it inaccessible. Since ObjectStore provided no way to re-load the database when this happened, a schema change had the same effect as a disk crash.

I-4. Transaction commits are time-consuming, possibly because of the time it takes to write the database to disk and re-acquire dropped locks.

I-5. ObjectStore provides a limited language-independent application program interface (API) through its interpretive query mechanism:[4] Only the "where" clause can be supplied at run time, it can invoke only a restricted class of functions, and it cannot, in general, invoke the operations associated with abstract data types stored in the database. See [Obj92], pages 184–191.

We were able to address these issues with a single cluster of closely related techniques.

The keystone of our solution was to implement MapBase as a multi-connection server that mediates between client requests and the underlying database. This server, from the point of view of the underlying database, is a single user. To avoid the need to implement our own locking and transaction roll-back, the server offers only single-statement transactions to its clients. The lack of multi-statement transactions is not ideal, of course, but we deem it to be an acceptable tradeoff at present; naturally we would prefer both good performance and multi-statement transactions.

To satisfy our requirement *R*-3, the need for a client/server architecture, the interface between application programs and the MapBase server is entirely text based. All commands and data are represented as character strings. Commands can be updates, retrievals, or control messages (such as set-date or commit-transaction), and usually take parameters. To put data into the

[3]ObjectStore Release 2.0 provides more support for schema evolution. See [Obj92], pages 265–323.

[4]The elegant ObjectStore query expressions described in [OHMS92] (and referred to as ObjectStore DML in [Obj92]) must be embedded in C++ and compiled.

database, an application program converts the data to text, and sends it as parameters to appropriate MapBase update commands. Retrievals are accomplished in two ways:

1. The MapBase server provides a simple, home-brew interpretive query language that allows select operations, count aggregates and sorting.

2. Retrievals that cannot be expressed in this language are coded in C++ and compiled into the Mapbase server.

In either case, the server responds with a stream of textual results, which the program converts to the desired internal representation.

With a single MapBase server interpreting text commands we were able to address issues *I*-1–*I*-5:

I'-1. Multi-user performance is improved because there is no lock contention: the MapBase server is the only process using the underlying database.

I'-2. The server supplies roll-forward recovery by logging all updates to a separate disk. The log is a logical log; it contains the update commands in their text representation.

I'-3. We evolve the schema by re-running all the roll-forward logs. If the schema change has invalidated any update statements in the logs, we must write an ad hoc program to appropriately revise the logs.

I'-4. The server amortizes the cost of commits by piggybacking them. The server performs a commit at approximately 10-minute intervals. Uncommitted updates can be recovered from the roll-forward log.

I'-5. The text commands and queries understood by the server constitute an API suitable for multi-lingual clients. Simple selection queries and `count` aggregates can be ad hoc. Other queries must be compiled into the MapBase server.

Figure 1 is a diagrammatic representation of MapBase's architecture.

Creating a server has allowed us to satisfy most of our key requirements. However, the limitations of our home-brew, ad-hoc query facility remain a problem, and we expect them to become more of a problem as we move into a new phase of our laboratory experimentation. These limitations have already cramped our design of the schema because we tend to make only those schema changes that can be easily accommodated by our home-brew query language. In addition, these limitations raise administrative difficulties when a query that cannot be expressed in our ad-hoc language requires the server to be recompiled.

In addition, our requirement *R*-2 for allowing MapBase to reflect changes in laboratory workflow is satisfied only with an uneasy compromise that still requires occasional database reloads for schema evolution.

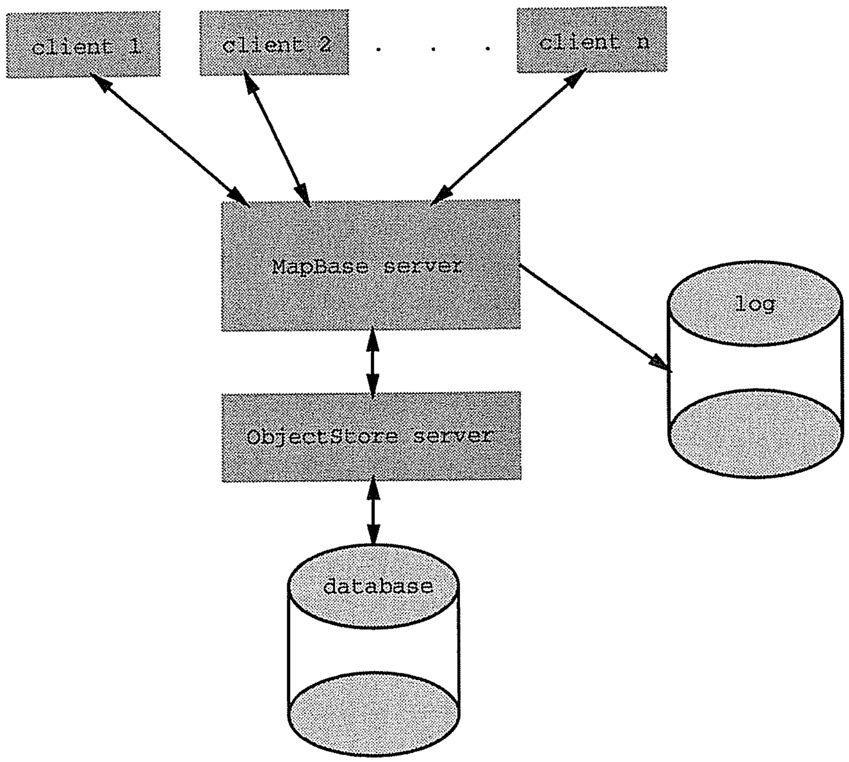


Figure 1: Architecture of the MapBase System.

3 What We Want From a Query Facility

In light of the continuing problems posed by the limitations of our ad hoc query facility, we would like to find a more powerful one that can be adapted to MapBase. It would also be desirable if such a facility could decrease the need to evolve the schema at the ObjectStore level. To try to understand what capabilities we want in a query facility, we consider, as examples, some queries that MapBase must answer.

3.1 DNA Sequence Operations

We represent DNA strands as sequences of the characters G, A, T, and C, that is, as strings in a restricted alphabet.[5] Each potential marker in MapBase is associated with a primary DNA sequence. MapBase also keeps track of many *sub*sequences of its primary DNA sequences, which it does by recording the length and starting position of the subsequence.[6] We wouldn't want to explicitly store these subsequences because of the possibility of introducing inconsistencies, and because we want to conserve space to keep the database

[5]Each letter denotes a particular constituent nucleotide. For more details on the biology please refer to e.g. [Lew90]

[6]For technical details please see [Goo93, GRS92].

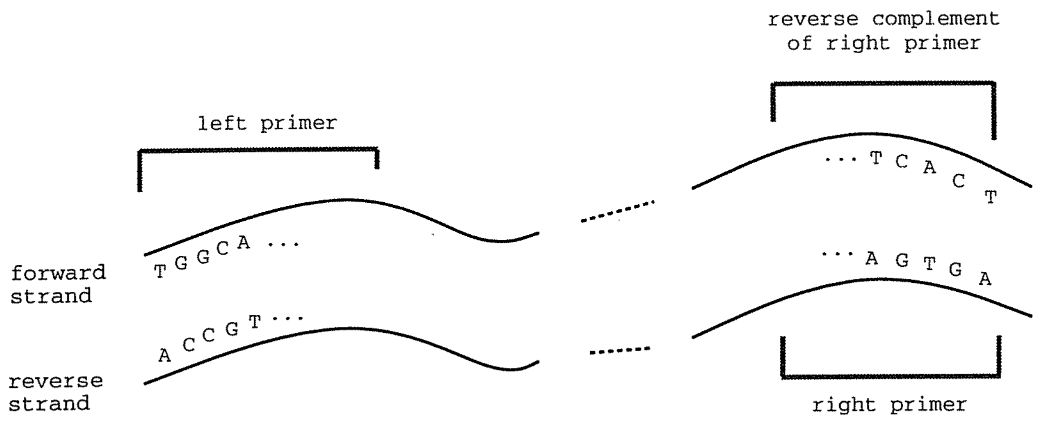


Figure 2: Left and Right Primers and Reverse Complementation.

The forward sequence is read left-to-right, and the reverse sequence is read right-to-left. Only the forward DNA sequence is stored in the database.

in physical memory. Therefore, a ubiquitous operation on DNA sequences is to take a subsequence of a DNA sequence.

Another common operation on DNA sequences is called "reverse complementation". A DNA molecule is a double helix consisting of two DNA strands (unpaired sequences). One strand is conventionally referred to as the "forward" strand, while the other is referred to as the "reverse" strand. (The reasons are not important here.) Given a forward sequence, the reverse complementation operator produces the matching reverse sequence, and vice versa. To do this, reverse complementation reverses the order of the sequence, and then substitutes the nucleotides according to the Watson-Crick base pairing rules: G$\leftrightarrow$C and T$\leftrightarrow$A. For example, the reverse complement of GATTCCGGG is CCCGGAATC. Naturally, when we store a primary DNA sequence, we store only one strand.

MapBase queries often require the reverse complement of a sequence. For example, in the laboratory mapping protocol, short DNA sequences are tagged at either end with even shorter sequences called "primers". The left primer is part of the forward sequence and the right primer is part of the reverse sequence. To find the left primer we need its starting position and length on this sequence; to get the right primer we use the starting position and length of the matching nucleotide sequence on the forward sequence, and compute the reverse complement of this subsequence (see figure 2).

For efficiency reasons, it is important that we be able to provide built-in predicates for ubiquitous operations like substring and reverse complementation.

3.2 Ready-For

A marker has a large number of boolean "ready-for" methods that keep track of its progress through the laboratory protocol. The methods are coded in C++ and linked into the MapBase server. For example, the steps for the early part of the analysis of a potential marker sequence are as follows:

1. Enter the sequence, S into MapBase.
2. Use a C-language client to duplicate-check S by comparing it to all sequences known to MapBase.
3. Use a (different) C-language client to check to see if any part of S closely resembles some sequence published in the national GenBank [BCC+91] database, in which case S might be associated with a known gene.

⋮

(many other steps)

⋮

The C-language client in step 3 invokes a program called BLAST, which runs on a National Center for Biotechnology Information computer[AGM+90]. One of the ready-for conditions is `ready_for_BLAST`. Each night the client mentioned in step 3, above, queries MapBase for all markers where `ready_for_BLAST` is true; for each such marker it retrieves from MapBase the DNA sequence needed as input by BLAST. `Ready_for_BLAST` is true for a marker, m, if either the following criteria are true:

- m has a sequence,
- m has been duplicate checked against previous sequences and no duplicates were found (i.e. at step 2), and
- m has never been examined by BLAST,

or

- m was previously examined by BLAST, but at an earlier date than the last duplicate-checker run (perhaps because the DNA sequence was revised by re-doing step 1).

This example illustrates both the pluses and minuses of the current design. On the plus side, ready-for relationships are easily represented in C++ as calculated marker attributes. But, on the minus side, this representation means that part of the laboratory protocol is hard-coded into the database server. Even a trivial change to the protocol, such as doing the BLAST search before the duplicate check, forces us to recompile and relink the server.

3.3 Map Position

In our mapping experiments, markers are first assigned to one of the chromosomes and then ordered relative to one another. Our mapping strategy[7] generates two types of orderings. The "framework" ordering involves markers that map unambiguously to a unique position. It is represented as an ordered list of markers with inter-marker distances. The "placement" ordering involves markers whose position is less certain. These markers are placed on the map by giving their distance relative to the nearest framework marker, and they can have multiple placements. Map position queries involve calculating the transitive closure over the framework and placement orderings. The query `?signed_distance("MPC101","MPC3003",D)`, (bind `D` to the signed distance between these two markers) must do the following:

1. If `MPC101` and `MPC3003` are on different chromosomes, or if one or the other isn't yet mapped, return an undefined result (the query is meaningless).

2. If they are both framework markers, find all intervening markers and add up the distances between them. If `MPC101` is above `MPC3003` on the chromosome, the sign of the result is positive; if `MPC3030` is above `MPC101` the sign is negative.

3. If one or both of the markers is a placed marker, find the nearest framework marker(s), calculate the distance between these two framework markers using rule (2), and adjust the distance by the placement to framework distance(s).

In MapBase this type of query is handled by an attribute called `map_position` which calls a C++ method which implements the rules given above. The `map_position` attribute is in turn used to answer queries involving the relative positioning of markers, the most frequent of which is to request all markers that map between a named pair of markers.

3.4 Pool Address

One of the types of mapping experiments performed in our laboratory is to determine which members of a set of approximately 25,000 large DNA segments called yeast artificial chromosomes (YACs) contain the small fragments we use as markers.[8] We cannot simply determine the sequence of each of these 25,000 segments because they are far too large; instead we use a simpler chemical test to determine if a YAC contains the marker sequence.

[7]Technically, here we are discussing mapping by "genetic linkage" analysis (see [Lew90]).

[8]Technically, here we are discussing "physical mapping", which uses methods different from genetic linkage analysis.

In order to avoid performing 25,000 individual experiments for each marker, the YACs are arrayed on a three dimensional grid and then pooled along the X, Y and Z coordinates to form a number of X, Y and Z coordinate pools. Each marker is then tested against each of the X, Y and Z pools. In a completely successful experiment, there is a single positive pool for each of the three coordinates, leading to a unique (X,Y,Z) address. However, due to experimental error there can be ambiguous or contradictory results: there may be a coordinate missing, such as (19:8:?), or there may be multiple positives for a given coordinate, such as (19:8:4,5).

In ambiguous cases the laboratory usually repeats the experiment. Sometimes this results in a unique address, but in other cases a different ambiguous result is obtained. Suppose, for example, that the first time an experiment is run the address (19:8:?) is determined, while on the repeat experiment, the address is (19:?:3). These two incomplete results can be merged to obtain a complete address, namely (19:8:3). MapBase handles ambiguous addresses by calculating the intersection of the vectors of all partial addresses known for the marker. This facility is implemented as a special-case query written in C++, since it cannot be handled in our limited ad-hoc query language.

3.5 Other Criteria

Expressiveness is not our only criterion for an ad-hoc query facility. Architectural issues, such as ease of adaptability to ObjectStore or another OODBMS, and the ease with which application programs can form queries and receive data are also important. And, of course, we must still satisfy our original requirements *R*-1–*R*-6.

Among these, schema evolution is still a problem in our current system. As mentioned above, MapBase's schema must continually evolve as the Genome Center revises its experimental work flow. For example, we order for each marker a quantity of the short DNA sequences called "primers" (discussed in 3.1). We receive 96 primers (left and right primers for 48 markers) in a box. A box has become a unit of work for a lab technician, so we recently started to record the association between marker and box.

In a deductive database, this change could be easily accommodated by adding a new base predicate `box(Marker_id,Box_id,Date,User)`, indicating that we received the primers for `Marker_id` in `Box_id`, with this information recorded on `Date` by `User`. Lest this seem trivial, consider that when we made this change in the current system, it required 108 lines of C++ code and required us to re-link the MapBase server.

4 Can We Use a Deductive Database?

The requirements for genome mapping pose several problems for the OODBMS we chose: we are hindered by the lack of an ad-hoc query language, poor multi-user performance, lack of schema evolution (at least in the original version), and the lack of roll-forward error recovery. We worked around these problems by creating a home-brew query language, a logging facility, and a multi-connection transaction server. However, in the process of working around these limitations, we discarded essentially all of the OODBMS's native transaction control and query facilities, and are using it as a persistent store server.

We are still hampered by the limitations of our ad-hoc query language and by difficulties in modifying the schema to accommodate rapid changes in the laboratory protocol. We hope to remedy these limitations in the next major revision of MapBase.

Several possible solutions have presented themselves:

- We considered trying to adapt an object SQL language such as CQL++ [DGJ92] or ZQL[C++] [Bla93] to MapBase. However, neither language has an interpreted implementation at present. There are two reasons that ad hoc queries require interpreted query execution rather than run-time compilation and loading:

  - It takes too long to compile even a short `.C` file (e.g. around 25 seconds on a Sparc 10, for a 73-line file plus headers), which is unacceptable in terms of our performance and response requirement, *R*-4.
  - An error in a dynamically loaded `.o` file can crash the MapBase server. It is unacceptable that a coding error in a single, ad hoc query make the database unavailable to all clients during recovery.

  One commercial company, Dharma, is trying to provide a portable, interpreted, object SQL [Dha], but it seems that a port to ObjectStore would be too much work for us.

- We could move to another OODBMS, such as VERSANT or O_2, that would provide fully general ad hoc queries and more flexibility in schema evolution.

- We could adapt a deductive, object-oriented query facility to our existing database. For example CORAL [RSS92, RSSS] and LDL [AOTZ, CGK89a, CGK89b, TZ86] were constructed in a way that might make it feasible for us to do this. Primary issues here would be performance and the ability add abstract data types and user-defined functions to the system (basically our requirements *R*-1 and *R*-4) above.

- We could move to a deductive, object-oriented database management system such as Aditi [VRK$^+$, VRK$^+$90], ConceptBase [Con], EKS-V1 [VBKL90], or Glue-Nail [DMP93] (or CORAL or LDL). Primary issues here would be, again, performance and the ability to add abstract data types and user-defined functions to the system. Additional considerations would be the robustness and data-administration facilities of the underlying storage manager.

A deductive, object-oriented query facility that supports user-defined predicates and abstract data types would be very attractive. We currently must compile queries such as the map position query into the MapBase server, but such queries would be directly expressible in a deductive query language (but not in a relational language). The need for user-defined predicates and abstract data types is illustrated by the subsequence and reverse complementation operations. For efficiency's sake we would want to provide these as built-ins.

There are also a number of "data mining" or "data dredging" [Tsu90] queries that we would like to pose. We hope to be able to write them using a deductive query language. Even if they prove to be too slow using the deductive language, it would still be worthwhile to have a system in which we could prototype such a query without writing hundreds of lines of C++ code. Once the query is prototyped, we could decide whether it was useful enough to justify a re-write in an imperative language.

We are currently reviewing a number of deductive database management systems and another OODBMS in an attempt to find a suitable platform on which to build the next version of MapBase.

Acknowledgments We thank Mary-Pat Reeve and Andre Marquis for their patient help in reviewing this paper.

References

[AGM$^+$90] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol. (England)*, 215(3):403–410, October 1990.

[AOTZ] Natraj Arni, Kayliang Ong, Shalom Tsur, and Carlo Zaniolo. The LDL++ system: Rationale, technology and applications. Unpublished manuscript, authors e-mail addresses: `{arni,ong}@mcc.com`, `zaniolo@cs.ucla.edu`.

[BCC$^+$91] C. Burks, M. Cassidy, M. J. Cinkosky, K. E. Cumella, P. Gilna, J. E.-D. Hayden, G. M. Keen, T. A. Kelley, M. Kelly, D. Krsitofferson, and J. Ryals. GenBank. *Nucleic Acids Research*, pages 2221–2225, 1991.

[Bla93] José A. Blakeley. ZQL[C++]: Extending the C++ language with an object query capability. In Won Kim, editor, *Database Challenges in the 1990's*. ACM Press/Addison-Wesley, 1993.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):65–77, October 1991.

[CGK89a] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Towards an open architecture for LDL. Technical Report ACA-ST-063-89, MCC, 1989.

[CGK89b] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Using modules and externals in LDL. Technical Report ACA-ST-036-89, MCC, February 1989.

[Con] ConceptBase Team, RWTH Aachen - Informatik V, Lehrstuhl Prof. Dr. Matthias Jarke, Ahornstr. 55 - 52056 Aachen, Germany. *ConceptBase V3.2 User Manual.* available by anonymous ftp from ftp.informatik.rwth-aachen.de (137.226.112.172), directory pub/CB/CB_3.2.

[DGJ92] S. Dar, N. H. Gehani, and H. V. Jagadish. A SQL for a C++ based object-oriented DBMS. In *Proceedings of the International Conference on Extending Database Technology*, March 1992.

[Dha] Opening up proprietary databases A white paper. Dharma Systems Inc.,15 Trafalger Square, Nashua NH 03063, USA.

[DMP93] Marcia A. Derr, Sinichi Morishita, and Geoffrey Phipps. Design and implementation of the Glue-Nail database system. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 147–156, May 1993.

[Goo93] Nat Goodman. Hints for effective design of object-oriented database application, 1993. Whitehead Institute for Biomedical Research.

[GRS92] Nat Goodman, Mary-Pat Reeve, and Lincoln Stein. The design of MapBase: An object oriented database for genome mapping, December 1992. Whitehead Institute for Biomedical Research.

[GST93] Susumu Goto, Norihiro Sakamoto, and Toshihisa Takagi. Object-oriented database with rule-based query interface for genomic computation. In *Proceedings of the Third International Symposium on Database Systems for Advanced Applications (Taejon, Korea)*, pages 65–72, 1993.

[LDL91] S. E. Lincoln, M. J. Daly, and E. S. Lander. PRIMER: a computer program for automatically selecting PCR primers, May 1991. Whitehead Institute for Biomedical Research.

[Lew90] Benjamin Lewin. *Genes IV.* Oxford University Press, 1990.

[LGA+87] E. S. Lander, P. Green, J. Abrahamson, A. Barlow, M. J. Daly, S. E. Lincoln, and L. Newberg. MAPMAKER: an interactive computer package for constructing genetic linkage maps. *Genomics*, 1(1):174–181, October 1987.

[MD92] D. McGoveran and C. J. Date. *A Guide to Sybase and SQL Server.* Addison-Wesley, 1992.

[$O_2$91] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, October 1991.

[Obj92] Object Design, Inc., One New England Executive Park, Burlington MA 01803, USA. *ObjectStore User Guide ObjectStore Release 2.0 for UNIX Systems*, October 1992. Number DU1002-100.

[OHMS92] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query processing in the ObjectStore database system. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 403–412, June 1992.

[ONT92] ONTOS, Inc., Three Burlington Woods, Burlington MA 01803, USA. *ONTOS DB 2.2 Developer's Guide*, February 1992.

[RSS92] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. An overview of CORAL. In *VLDB*, August 1992. A longer version is available from the authors.

[RSSS] Raghu Ramakrishnan, Praveen Seshardi, Divesh Srivastava, and S. Sudarshan. *The CORAL User Manual A Tutorial Introduction to CORAL.* Computer Sciences Department, University of Wisconsin, Madison WI 53706, USA. Available by anonymous ftp from `ftp.cs.wisc.edu`.

[SGT] Norihiro Sakamoto, Susumu Goto, and Toshihisa Takagi. A deductive database system for analyzing human nucleotide sequence data. *The International Journal of Bio-Medical Computing*, In press.

[Tsu90] Shalom Tsur. Data dredging. *Data Engineering*, 13(4), December 1990.

[TZ86] Shalom Tsur and Carlo Zaniolo. LDL: a logic-based data language. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 33–41, August 1986.

[VBKL90] Laurent Vieille, P. Bayer, V. Küchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *AAAI Workshop on Knowledge Base Management Systems*, 1990.

[Ver93] Versant Object Technology Corporation, 4500 Bohannon Drive, Menlo Park CA 94025, USA. *VERSANT Object Database Management System Release 2 System Manual*, July 1993.

[VRK$^+$] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, Peter J. Stuckey, Tim S. Leask, and James Harland. An introduction to the Aditi deductive database system.

[VRK+90] Jayen Vaghani, Kotagiri Ramamohanarao, David B. Kemp, Zoltan Somogyi, and Peter J. Stuckey. The Aditi declarative database system. In J. Chomicki, editor, *Proceedings of the NACLP'90 Workshop on Deductive Databases*, 1990.

MIMSY: A System For Analyzing Time Series Data in the Stock Market Domain

William G. Roth*
Tandem Computers, Cupertino, CA.

Raghu Ramakrishnan
University of Wisconsin, Madison

Praveen Seshadri †
University of Wisconsin, Madison

Abstract

This paper describes a real–world application built on top of the CORAL deductive database system. This application is meant to demonstrate the power of CORAL not only as a deductive database but also as a generic extensible database system. Mimsy is a stock market historical reporting system that can answer questions about daily stock market pricing data. The paper describes the Mimsy system, and issues related to its design and implementation.

1 Overview

CORAL [RSS92] is an extensible deductive database system developed at the University of Wisconsin. While providing all the functionality of a logic programming environment, CORAL also provides the functionality of a general–purpose extensible database system [RSSS93]. The system includes an interpreter capable of processing a declarative rule–based language and an imperative environment where CORAL can be accessed from within a host language(C++ [ES90]). This environment is geared toward the building of non–trivial database applications.

Mimsy is a system for asking questions about stock market data, and has been built on top of CORAL. The goal is to provide an SQL–like language specifically geared toward the application of querying stock market data. Mimsy is inspired by a commercial application sold by Logical Information Machines, Inc., called MIM[1] [Lew92][2]. Like MIM, Mimsy allows the user to execute ad hoc queries against the stock market data. It should be noted that Mimsy is at once both more powerful and less powerful than MIM. For example, MIM has an extensive set of date primitives, whereas Mimsy has only a rudimentary

The work of the authors was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563.

*The work of this author was performed while he was at the University of Wisconsin, Madison, and was partially supported by the grants listed above.

†The addresses of the authors are Tandem Computers, 10100 Tantau Avenue, Mail Stop 251-04, CA 95014, USA, and Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA. The authors email addresses are roth_bill@tandem.com, raghu@cs.wisc.edu and praveen@cs.wisc.edu.

[1]Logical Information Machines and MIM are registered trademarks of Logical Information Machines, Inc.

[2]"Mimsy" is meant to be thought of as an adjective meaning MIM–like.

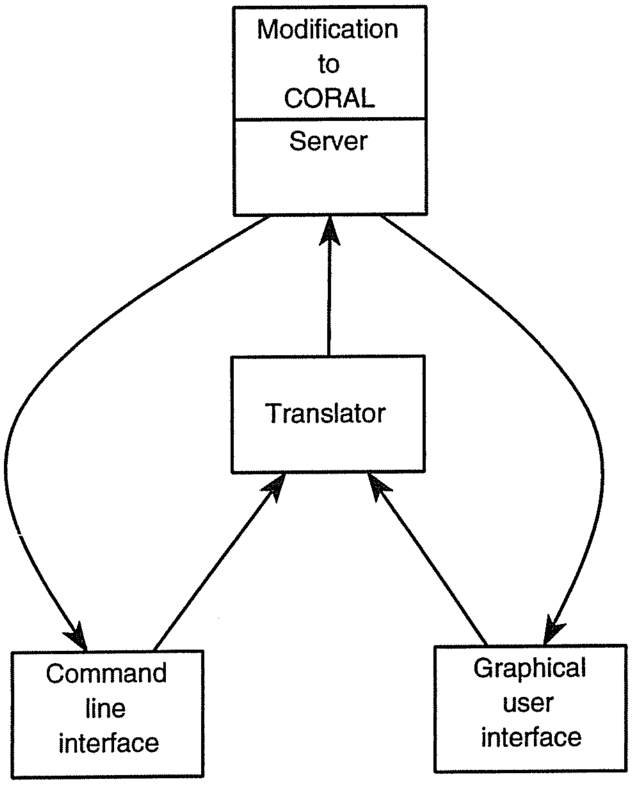


Figure 1: The design of the Mimsy system

subset. However, MIM's range of extensibility is rather limited, while Mimsy's extensibility has the full range of expressiveness of declarative CORAL, by virtue of the fact that it is built on top of CORAL.

Mimsy consists of three logical components.

- an interface that accepts queries in the form of the Mimsy query language.
- a translator that converts the Mimsy query into corresponding CORAL queries.
- the Mimsy server that processes the CORAL queries and returns the answer back to the interface for display to the user.

A command line interface is provided for interactive querying, and a graphical point-and-click interface for composing queries is also available. Utilities for sorting and graphing the results of queries are provided. This paper, however, focuses on the Mimsy language and implementation issues. The interface is described in detail in [Rot93].

The following points are worth emphasizing:

1. Mimsy can deal with significant amounts of data and quite complex queries, providing interactive responsive time. The recursive query capabilities of CORAL are necessary for expressing many natural concepts (e.g., a "bull market", "consecutive peaks", etc.) in this domain.
2. It demonstrates the power of CORAL's extensibility features.

3. It illustrates the use of CORAL in a client-server mode.
4. It is a substantial application that was completed primarily by one graduate student over one semester. We believe that this is considerably less than the effort that would be involved in building a similar package from scratch without the use of a system such as CORAL.

2 The Language

The basic structure of a Mimsy query is as follows:

```
select <something>
when <something-else>
save as <something-to-be-saved>
```

Both the `when` and the `save as` part of the query are optional. The `select` clause determines what data will be returned to the user. The `when` clause determines for which dates the data in the `select` clause will be shown. The `save as` clause specifies that the answers should be stored in a relation instead of displaying the results of the query on the screen. We discuss how the saved relation can be used in subsequent Mimsy queries in Section 6.2.

The data operated on by Mimsy is called a *series*. A series is a vector of price data. A series can be thought of as a binary relation with the first column specifying the date or time index, and the second column specifying the value of the series at that index. In Mimsy, a series is identified by its ticker symbol and an identifying attribute. For example, `close of abc` represents the close series for the stock whose ticker symbol is `abc`.

Operations, known as *aggregates*, can be applied to series to produce values. Aggregates and a time range are applied to base series. For example `the 30 day average of close of abc` represents the 30 day moving average of the stock whose ticker symbol is abc.

2.1 The Select Clause

In the Mimsy language the `select` portion of the query is comprised of one or more select attributes separated by commas. A select attribute is composed of a select expression followed by an optional `repeated` clause. A select expression is a series, aggregate on a series, or an arithmetic expression involving a series or an aggregate. Each select expression can be offset by a time period. Examples of the select clause are shown below, that demonstrate the use of the `repeat` clause and the `offset` clause.

```
select close of ibm;
select 3 * close of ibm repeated for 10 days;
select the 4 day average of close of ibm + 1;
select close of ibm - close of ibm 1 day ago
    repeated from the previous 2 days to
    the next 3 days;
```

2.2 The When Clause

The when clause is a list of predicates, separated by logical connectives that choose the dates for which the data in the select clause is to be shown. The when clause comes in 3 flavors: relational operator clauses, change clauses, and crosses clauses. All of these clauses can be further modified by a condition interval, which is a date range for which the condition must hold true.

The relational operator clause tests one select expression against another for a specific date. For example:

```
select close of abc when close of abc > 12;
select close of abc when close of def > close of b;
select close of abc when close of abc > close of b * 6;
```

For these queries, the implication is that the condition holds over 1 day. For a longer date range, the condition interval can be used. For example:

```
select close of xrx when close of xrx > 12 over 3 days;
```

The change clause tests whether a select expression is up or down over some period.

```
select close of abc when close of abc is up at least 10;
select close of abc when close of abc is down more than close of b;
select close of abc when close of abc is up more than 10%;
select close of abc when close of x is not down;
```

The crosses clause tests when one series crosses another. A cross occurs when, for example, one stock's price moves above the price of another when the first stock was smaller or equal in price to the second stock in the previous time period. For example:

```
select close of abc when close of abc crosses close of b;
```

This query will select the close of abc on all dates when the close of abc crosses above or below the close of b. The query can also specify either "above" or "below" in the crosses clause to specify which type of cross is of interest. For example:

```
select close of abc
when close of abc crosses above
    the 30 day average of close of abc;
```

2.3 Aggregates

Aggregates compute a single answer from the time range and series that are given as arguments. The aggregates included in the system are shown in Figure 1. The average aggregate computes a moving average of a series for the time period specified. The definition of min, max and sum are the same as in SQL. Move represents the total change in price of the series from the beginning of the time period to the end of the time period. Pcmove represents the total percentage change in price of the series from the beginning of the time period to the end of the time period.

All aggregates are used as follows, using average as an example:

#	Name	Mimsy Aggregate name
1	Average	average
2	Max	max
3	Min	min
4	Sum	sum
5	Move	move
6	Percentage Move	pcmove

Table 1: Mimsy aggregates

```
select the 5 day average of close of ibm
when the 3 day average of ibm is up more than 10%;
```

This query will show the 5 day moving average of the close of IBM on each day when the 3 day average rises by more than 10 percent in 1 day.

3 The Translator

Once a query has been received by either the command line interface or the graphical interface, the string is sent to the translator. If the query string acquired from the interface parses correctly, it is translated into CORAL commands and sent to the server over a socket connection. The result of the query, when received from the server, is read from the socket and sorted, and then displayed either textually or graphically.

The overall task of the translator is simple. It generates CORAL rules for the `when` clause so that all its results are collected into one temporary relation. This relation corresponds to the list of all dates that satisfied all the conditions in the `when` clause. These are the dates for which the `select` clauses will be evaluated. Separate abstract syntax trees are genarated for the `select` and `when` clauses, and are subject to various transformations. These transformations handle date ranges, and boolean expressions. Optimizations to handle aggregates and REPEAT clauses are also performed. Finally, the CORAL code corresponding to the Mimsy query is generated and sent to the server. The details of the translation are fairly complex and are described in [Rot93].

4 Series Data

There are currently 10 base series each defined for 165 stocks going back 5 years. The data for these series comes from the University of Chicago's Center for Research in Security Prices (CRSP). We have actually used a small fraction of the available data, which covers over 1600 stocks, going back over 20 years. It is important to note that CORAL only brings the data relevant to stocks in a given query into main memory; this loading is done very efficiently. Considering the entire data set supplied by CRSP will not affect performance significantly.

The series and their keywords in Mimsy are shown in Table 2. Series 1–5 are base series. That is, they are are actual values describing some property of the stock. Series 6–10 are composite series that represent some kind calculation performed by CRSP on the base data. These composite series are pre–calculated and stored as base data.

#	Name	Mimsy Sequence name
1	Close	close
2	High	high
3	Low	low
4	Volume	volume
5	Shares outstanding	shares
6	Beta	beta
7	Beta return	betaret
8	Capitalization	cap
9	Return	ret
10	Standard deviation	stdev

Table 2: Base series defined in Mimsy

5 The Server

The server is an iterative, connection–oriented server constructed as an extension to the basic CORAL system. Operationally, the server front–end merely listens to a socket, treats whatever comes over the socket as commands to the CORAL system, and returns the result back to the originator. However the underlying CORAL database engine has been extended to perform efficiently in this application domain. A new type of relation, called a *fast array relation*, has been added to support stock data, and all of the series aggregates have been coded directly as CORAL "built–in" relations.

The server first loads the date translation table, which translates regular dates into date integers, and the CORAL program to handle the date offsets. Next, the server loads in the relation that defines the catalog of all valid series, and initializes the corresponding relations. The server then creates the `dates` relation, which is a unary relation that contains a date identifier for every valid date in the system[3]. In the next step, the server loads in semantic information that will be sent to and used by both the command line interface and the graphical interface. These catalogs include the currently defined series, properties of series and aggregate definitions.

5.1 Fast Array Relations

One of the classic problems of dealing with stock market data is that it does not fit very well into the relational model. In the relational data model, every "object" is treated as a table of an unspecified (though potentially large) number of columns. The cost of maintaining this generality often makes accessing stock market, and indeed all time–series, both space and time inefficient. Some Wall Street investment banks have even been known to use environments like APL [GR84, Ive62] that are geared toward dealing with vectors and matrices, and eschew the use of RDBMSs completely. Since CORAL is a general purpose deductive database system, using the basic system would result in similar inefficiencies.

In order to allow CORAL to operate efficiently on time–series data, a new type of relation has been added to CORAL which facilitates fast access of sequence data. *FastArrayRelation* is a subclass of the CORAL Relation class and was added to the CORAL class hierarchy using the extensibility features provided by the system. A filename is passed to the FastArrayRelation when it is created specifying the location of the data file. The data is loaded into the file the first time the relation is iterated over. The data is stored in the file as a binary array, so that it can be loaded directly into memory without any translation. It is assumed that the data in the file is in the host machine's numeric format.

[3]Note that the stock trading dates are not sequentially increasing. Holidays and weekends interrupt the sequentiality.

The first column is an integer index. The second column is the value for that index, and is either a double precision floating point number or an integer. The first column of the relation is not actually stored. When the data is loaded, the file header information contains the number of elements in the array and the lower bound. Using this information, the first column can be synthesized. It is assumed that FastArrayRelations are read–only, since the stock data is relatively static.[4] The second column is retrieved by a simple array access.

5.2 Aggregates

Mimsy aggregates are written to recognize opportunities for efficient execution. This is done by having the aggregate cache its arguments and some of its computations from the previous invocation. Note that forward motion of time is always assumed when an aggregate is being iterated across a range of dates. For example, if the `average` aggregate is called on day 1 through day 20 of a series, it will cache the number of days in the range, the name of the portfolio or series, and the sum of the values over the range. If the next invocation is for day 2 through day 21 of the series, to compute the average it is sufficient to get the value for the previous lower bound of the range, subtract it from the sum, get the value for new upper bound for the range, add it to the sum and divide by the number of days in the series. If the next invocation is not for the next day in the window, the cached values are discarded and the computation must proceed as if called on the first day of a range. The aggregates `min` and `max` are optimized similarly. The effect of having aggregates optimized in this way is that the aggregates will recognize many opportunities for optimizations. This obviates the need for any higher level optimizations other than sorting the data before the aggregates are executed.

6 Extensibility

Mimsy utilizes and demonstrates the extensibility of the underlying CORAL system. However, Mimsy itself is also extensible at all levels, so that it is possible to enhance the system with more data, further functionality and even greater query expressivity. This section discusses more how extensions to the query language are handled.

6.1 Extending the Server

Users can define a new series in one of two ways.

Data can be added to the server by placing a properly formatted file in the data directory. Tools are provided to appropriately format the data. To add a derived series to the server via a CORAL module, the module should be loaded in by the server during startup. In both cases, information about the new schema should be added to the server catalogs.

Definitions of new aggregates are added to the server in a similar fashion. An aggregate is defined as follows, using "average" as an example:

```
seqaverage(BeginDate,EndDate,Series,Answer).
```

[4] A sub-class of FastArrayRelations that allows inserts too is provided for the case of a time–series that needs to be saved permanently, or temporarily.

The aggregate computes the appropriate value from `BeginDate` to `EndDate` for the series or portfolio and returns the result in `Answer`. There are two ways to define new aggregates, as new built-ins to be compiled into the server, or as CORAL modules to be loaded into the server at startup. The catalog of aggregate definitions needs to be updated to include the new aggregate. Some of these aggregate definitions may involve the use of recursion (for example, a moving average), and the ability of CORAL to specify recursive rules is useful in this regard.

6.2 Extending the Language

It is likely that a user will want to pose some queries not readily expressible in the Mimsy language, but that can be expressed in the underlying CORAL language. For instance, in the stock market domain, there are a variety of queries that are recursive in nature (for example, determining maximal montonic trends, or local minima and maxima). These queries are easily expressible in CORAL since it supports recursive rules(the appendix has a couple of examples of such queries). We provide a way to define CORAL predicates and extend the Mimsy language to make these new definitions accessible; this *extensibility* allows a sophisticated user to customize the language. Indeed, this is also the mechanism used to refer to relations produced by the *save as* clause (see Section 2).

Extensibility in the Mimsy language is provided by allowing strings found at certain places in the grammar to be passed uninterpreted to the server. Also, variables in a query that refer to variables in extensibility strings are allowed to appear in `select` expressions. A special variable, "`Date`", can appear in the argument string. In the translation, the string `Date` will be replaced by the variable that represents the current date under consideration. For example, suppose we had a predicate "extend" that returned some value. For example,

```
select close of abc,"new_series(Date,Val)",Val
when "extend(Date,X)" and X > 12.
```

would be translated into:

```
whendates(D) += dates(D),extend(D,X),X>12.
?whendates(D),a_close(D,D0),new_series(D,Val),D1=Val,
   dateuntranslate(D,X),print(X,D0,D1).
```

The extensibility strings can contain any valid CORAL code, and are interpreted to a very limited extent by the Mimsy front-end. Some knowledge of the translation process of Mimsy to CORAL is therefore necessary to effectively use this feature.

7 Conclusion

In developing Mimsy, we have tried to demonstrate that CORAL provides support for realistic database applications. The extensibility features of CORAL are heavily utilized to ensure efficient performance. The design of the Mimsy system itself involved some interesting language issues and implementation decisions. While we have touched upon them briefly, they are discussed in detail in [Rot93]. The system is currently being extended to handle portfolio queries, requiring significant extensions both of the language and the implementation.

The experience of developing Mimsy offers evidence of the power of deductive systems. The total programming effort was about two man-months, of which about one month was spent on developing the graphical interface. The actual design of Mimsy and acquiring familiarity with CORAL took about a month. We note that the primary implementor of Mimsy, W.G. Roth, was relatively unfamiliar with many of the implementation details, such as relation interfaces, when he began this project. While he did have some familiarity with aspects of CORAL, this was probably offset by the fact that the C++ interface was in a state of flux during the course of this project. In any case, while such considerations are hard to quantify, it is fair to note that Mimsy was designed and developed in about a semester primarily by one programmer.

The CORAL system provided considerable support. First, the C++ interface facilitated the definition of a number of functions needed to support the Mimsy language. Second, the extensibility of CORAL was used to represent series data effficiently while retaining a relational view. Third, the ability to pose ad-hoc queries relies upon CORAL to parse and evaluate rules; this allowed us to focus on high-level issues relevant to Mimsy rather than details of query processing. Finally, the full power of CORAL rules can be used to define additional functions to add to the power of Mimsy, and some of the pre-defined functions are indeed defined in this way.

8 Appendix

CORAL rules defining the N-day Moving Average of a Series

```
% These rules use the moving average for the previous period to
% incrementally compute the moving average for the next period.

% series(SeriesName, Date, Value)
% movavg(SeriesName, NoOfDays, Date, Value)

movavg(Sname,N,Day1,A) :-
        t2(Sname,N,Day1,N,V), A=V/N.

movavg(Sname,N,D1+1,A) :-
        movavg(Sname,N,D1,A1),
        series(Sname,D1,Old),
        series(Sname,D1+N,New), A=A1+(New-Old)/N.

% t2 just computes the average for the first N days.

t2(Sname,N,Day1,1,V) :-
        from(Day1),
        series(Sname,Day1,V).

t2(Sname,N,D,M1+1,V1+V2) :-
        t2(Sname,N,D,M1,V1), M1<N,
        series(Sname,D+M1,V2).
```

CORAL rules defining the Runs(maximal monotonic trends) of a Series

```
% This program analyses a series and generates a result series
% that describes maximal decreasing/non-decreasing "runs" in the
% original series.

run(Sname,Date,Dir,max(<L>)) :- trend(Sname,Date,Dir,L).

% for each date, trend computes the longest current inc/dec (possibly
% non-maximal) "run"

% the following rules deal with the case when the "observed trend" (inc/dec)
% continues on the current date.

trend(Sname,Date,nondec,L+1) :-
        trend(Sname,Date,nondec,L), series(Sname,Date+L,V1),
        series(Sname,Date+L+1,V2), V2>=V1.

trend(Sname,Date,dec,L+1) :-
        trend(Sname,Date,dec,L), series(Sname,Date+L,V1),
        series(Sname,Date+L+1,V2), V2<V1.

% the following rules deal with a "change in direction"

trend(Sname,D1,nondec,1) :-
        trend(Sname,Date,dec,L), series(Sname,Date+L,V1),
        series(Sname,Date+L+1,V2), V2>=V1.

trend(Sname,D1,dec,1) :-
        trend(Sname,Date,nondec,L), series(Sname,Date+L,V1),
        series(Sname,Date+L+1,V2), V2<V1.

trend(Sname,Day1,nondec,1) :-
        from(Day1), series(Sname,Day1,V1), series(Sname,Day1+1,V2), V2>=V1.

trend(Sname,Day1,dec,1) :-
        from(Day1), series(Sname,Day1,V1), series(Sname,Day1+1,V2), V2<V1.
```

References

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusett, 1990.

[GR84] Leonard Gilman and Allen J. Rose. *APL: An interactive approach.* John Wiley & Sons Inc., New York, third edition, 1984.

[Ive62] Kenneth E. Iverson. *A programming language.* John Wiley & Sons Inc., New York, 1962.

[Lew92] Peter H. Lewis. A Fast Way to Discover Patterns in Vast Amounts of Data. *The New York Times*, pages 16–17, August 23 1992.

[Rot93] William Gibson Roth. MIMSY:A System for Analysing TimeSeries Data in the Stock Market Domain. Master's Thesis, University of Wisconsin, Madison, 1993.

[RSS92] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[RSSS93] Raghu Ramakrishnan, Divesh Srivastava, S.Sudarshan, and Praveen Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1993.

Efficient Visual Queries for Deductive Databases

Dimitra Vista
Department of Computer Science
University of Toronto
Toronto, Canada M5S 1A4

Peter Wood*
Department of Computer Science
University of Cape Town
Rondebosch, South Africa 7700

Abstract

We report on the efficiency of queries formulated with the Hy$^+$/GraphLog data visualization system, a substantial application which utilizes the CORAL deductive database system. Hy$^+$ itself has been used in a number of application areas, including software engineering, network management, and the debugging of distributed programs. We focus on the translation of GraphLog queries to CORAL programs as well as the performance of the resultant programs on large data sets. One source of inefficiency in programs involving recursion and query constants is that CORAL is not able to detect that the programs are factorable. We provide an alternative translation which always propagates constants to base relations, with a consequent improvement in performance.

1. Introduction

Hy$^+$ is a data visualization system under development at the University of Toronto. In Hy$^+$, data can be visualized as a hygraph (a hybrid between hypergraphs and higraphs),

*This work was done while the author was visiting the Computer Systems Research Institute, University of Toronto, Toronto, Canada M5S 1A1.

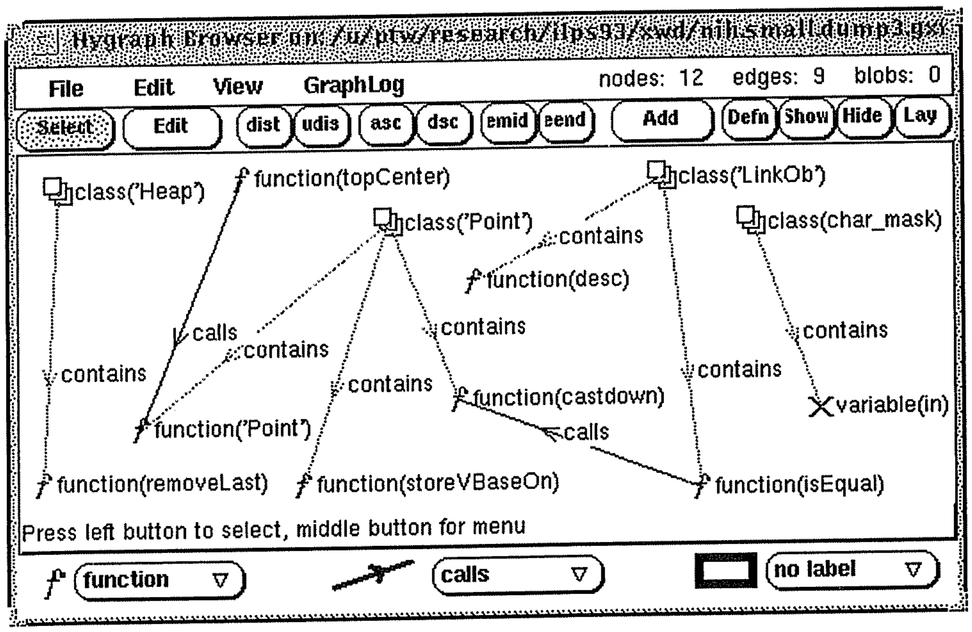


Figure 1: Visualizing a portion of the NIH class library.

on which queries filtering out information or defining new relationships can be formulated. These queries are expressed in GraphLog, itself a visual query language. The system has been used successfully in applications such as software design [6], network management [2], and debugging of distributed programs [3].

Query processing in Hy$^+$ is performed by translating queries (and data, if necessary) into logic programs suitable for execution by one of three backends: CORAL, LDL, or Prolog. In this paper, we concentrate on the efficiency of queries executed on the CORAL backend [13]. Details of the translation to Prolog can be found in [8], while that for LDL is described in [7]. In addition, we do not address the features of negation and aggregation which GraphLog also provides [7].

As an example, Figure 1 shows a portion of the structure of the NIH public domain C++ class library, visualized as a labelled, directed graph. The objects of interest, represented by nodes, are classes, functions and variables. Relationships include the subclass and friend relationships between classes, the contains relationship between a class and its member functions, a reference relationship between functions and variables, and a calls relationship between functions.

A GraphLog query on the graph of Figure 1 is shown in Figure 2. The query first defines a relationship *depends* between classes, and then asks for the transitive closure of this relationship, restricted to those classes which depend directly or indirectly on the

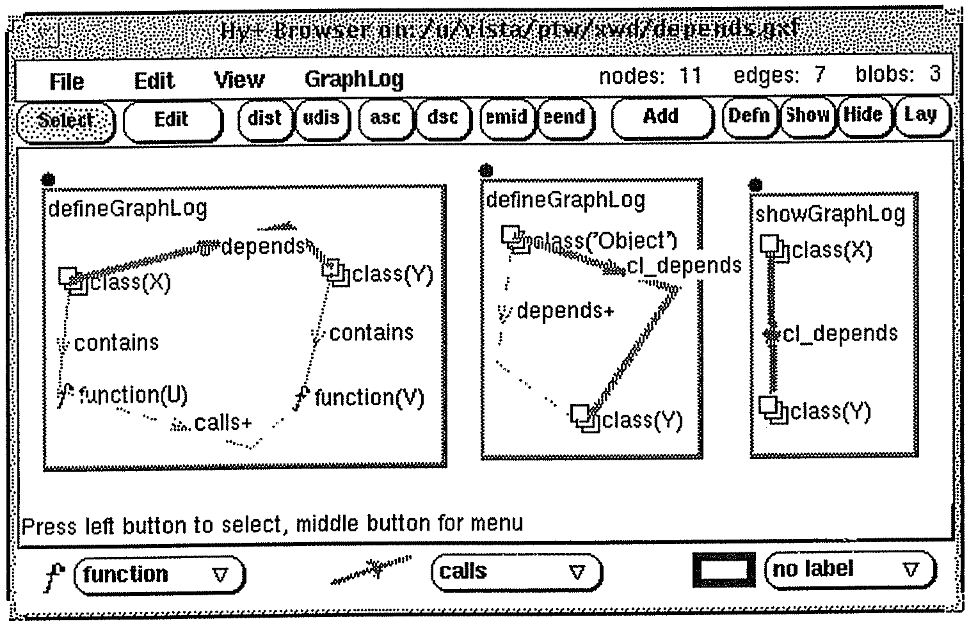


Figure 2: A GraphLog query on the database of Figure 1.

class "Object." Class X depends on class Y if X contains a member function which calls directly or indirectly a member function of Y.

The efficiency of evaluating queries like that of Figure 2 is affected significantly by the choice and sophistication of the translation from GraphLog to CORAL. For example, with a careless ordering of subgoals by the present GraphLog translation, the query can take over 3.5 minutes to execute on a database of 9124 tuples, even using the `@factoring` annotation provided by CORAL. With the correct ordering of subgoals, this can be reduced to around 31 seconds. In this paper, we provide an alternative translation for GraphLog which reduces the execution time, in this case, to under 1.5 seconds.

Although our emphasis is on translating GraphLog queries, the techniques we develop can be applied independently. In addition, we comment on other issues such as subgoal ordering and the efficiency of various annotations with which one can control the evaluation mechanism chosen by CORAL.

The outline of the rest of the paper is as follows. In the next section, we give a brief description of the GraphLog query language, along with the current technique for translating to CORAL. Section 3 is devoted to describing the alternative translation scheme, while Section 4 provides a comparison of the efficiency of the two techniques, using a number of sample queries and databases. Conclusions and directions for further work are discussed in Section 5.

2. The GraphLog Visual Query Language

GraphLog is a graphical formalism for visual manipulations of database visualizations [5]. It is suitable for applications of database technology where the data has a graph-like structure which can be exploited to provide visual presentations of the data [4, 6].

2.1. Syntax and Semantics

GraphLog visualizations are based on the notion of hygraphs. Hygraphs are directed labelled graphs which, in addition to labelled nodes and edges, also contain labelled *blobs*. A *blob* in a hygraph generalizes the concept of an edge: it is a relation from a node, called the *container* node, to a set of other nodes, called the *contained* nodes. It can be used to group similar objects together. Visually, it is represented as a rectangular box associated with the container node. All contained nodes are displayed inside the container's box.

In GraphLog, a *term* is either a constant, a variable, an anonymous variable (as in Prolog), or a function f applied to a number of terms. An *edge* or *blob label* is an expression generated by the following grammar, where $\bar{T}$ is a sequence of terms and p is a predicate:

$$E \rightarrow E|E; E.E; -E; \neg E; (E); E+; E*; p(\bar{T})$$

This is essentially a grammar for regular expressions, with "|" representing alternation, "." concatenation, "–" inversion (a traversal from head to tail, rather than tail to head), "¬" negation (the absence of a path), "+" transitive closure, and "∗" reflexive transitive closure.

Database instances are hygraphs whose nodes are labelled with ground terms and whose edges and blobs are labelled with ground predicates. Database instances of the object-oriented or relational model can easily be visualized as hygraphs. For example, an edge (blob) labelled $p(\bar{X})$ from a node labelled T_1 to a node (containing a node) labelled T_2, corresponds to tuple $(T_1, T_2, \bar{X})$ of relation p in the relational model.

Queries are hygraphs whose nodes are labelled by terms and each edge (blob) is labelled by an edge (blob) label. There are two types of queries: *define* and *filter*. In both types, the query hygraph represents a pattern; the query evaluator searches the database hygraph for all occurrences of that pattern. The difference between the two types of queries stems from their interpretation of *distinguished edges*, explained below.

A *define* query must have one *distinguished* edge or blob, labelled by a positive literal.

The meaning of a *define query* is to define the predicate in this distinguished literal in terms of the rest of the pattern. The semantics of *define* queries is given by a translation to stratified Datalog. Each *define* hygraph G translates to a rule with the label of the distinguished edge or blob in the head, and as many literals in the body as there are non-distinguished edges and blobs in G. If an edge is labelled with a regular expression, additional rules to define its predicate are necessary.

A *filter* query can have several distinguished edges and blobs; they represent those objects that the user wishes to see. The query evaluator searches the database for all occurrences of the pattern, and retains the objects that match the distinguished edges and blobs from each occurrence. Edges and blobs defined by *define* queries can of course be used in *filter* queries.

2.2. The Translation

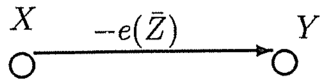
As explained above, the meaning of a *define query* in GraphLog is to define the predicate in the distinguished literal of the query. Suppose between nodes labelled X and Y, there is an edge labelled $p(\bar{Z})$, where X, Y are terms and $\bar{Z}$ is a sequence of terms. We will denote this by $p(X, Y, \bar{Z})$. In fact, the meaning of such an edge is given by the Datalog fact $p(X, Y, \bar{Z})$.

Given a *define* GraphLog query Q with non-distinguished edges $p_1(\bar{X}_1), \ldots, p_n(\bar{X}_n)$ the translation for Q contains the following rule that defines the predicate p of the distinguished edge $p(\bar{X})$:

$$p(\bar{X}) : - \ l_1(\bar{F}_1), l_2(\bar{F}_2), \ldots, l_n(\bar{F}_n)^1.$$

where $l_i(\bar{F}_i)$ is $p_i(\bar{X}_i)$, if the edge is positive, $\neg p_i(\bar{X}_i)$, if the edge is negated, or it is defined recursively on the structure of the label according to the following algorithm:

- Inversion



corresponds to the call $e(Y, X, \bar{Z})$.

[1]The idea of using $\bar{F}_i$ instead of $\bar{X}_i$ is that constants and anonymous variables need not appear in the call of a predicate that is associated with an expression (defined with additional rules).

- Concatenation

$$\begin{tikzcd} X \arrow[r, "e_1(\bar{Z}_1).e_2(\bar{Z}_2)"] & Y \end{tikzcd}$$

$$conc(X,Y,\bar{F}_1,\bar{F}_2) :- e_1(X,T,\bar{Z}_1), e_2(T,Y,\bar{Z}_2).$$

where $\bar{F}_i$ contains the named free variables of $\bar{Z}_i$ (bound terms and anonymous variables are not included), and T is a variable appearing nowhere else in the rule. Note that if a variable appears in both Z_1 and Z_2, then it will be carried twice in the head.

- Alternation

$$\begin{tikzcd} X \arrow[r, "e_1(\bar{Z}_1)|e_2(\bar{Z}_2)"] & Y \end{tikzcd}$$

$$alter(X,Y,\bar{F}_1,\bar{F}_2) :- e_1(X,Y,\bar{Z}_1).$$
$$alter(X,Y,\bar{F}_1,\bar{F}_2) :- e_2(X,Y,\bar{Z}_2).$$

where $\bar{F}_i$ contains the named free variables of $\bar{Z}_i$. Note that if a variable appears in both Z_1 and Z_2, then it will be carried twice in the head.

- Closure

$$\begin{tikzcd} X \arrow[r, "e(\bar{Z})+"] & Y \end{tikzcd}$$

$$tc\_e(X,Y,\bar{F}) :- e(X,Y,\bar{Z}).$$
$$tc\_e(X,Y,\bar{F}) :- e(X,T,\bar{Z}), tc\_e(T,Y,\bar{F})).$$

where $\bar{F}$ contains the named free variables of $\bar{Z}$, and T is a variable appearing nowhere else in the rule.

- Kleene Closure

$$\begin{tikzcd} X \arrow[r, "e(\bar{Z})*"] & Y \end{tikzcd}$$

$$kleene\_e(X,X,\bar{F}) :- e(X,Y,\bar{Z}).$$
$$kleene\_e(Y,Y,\bar{F}) :- e(X,Y,\bar{Z}).$$
$$kleene\_e(X,Y,\bar{F}) \quad :$$
$$- e(X,T,\bar{Z}), kleene\_e(T,Y,\bar{F}).$$

where $\bar{F}$ contains the named free variables of $\bar{Z}$, and T is a variable appearing nowhere else in the rule.

Since, for the purposes of this paper, we assume only trivial filter queries, we will not describe their translation here. As an example, the query of Figure 2 is translated to the following Datalog program (we have omitted the full module definition for CORAL):

```
cl_depends(class("Object"), class(Y)) :-
                        tc_depends(class("Object"), class(Y)).
tc_depends(X, Y) :- depends(X, Y).
tc_depends(X, Y) :- depends(X, Z), tc_depends(Z, Y).
depends(class(X), class(Y)) :- tc_calls(function(U), function(V)),
                               contains(class(Y), function(V)),
                               contains(class(X), function(U)).
tc_calls(X, Y) :- calls(X, Y).
tc_calls(X, Y) :- calls(X, Z), tc_calls(Z, Y).
```

Note, in the fourth rule, how the (non-deterministic) ordering of subgoals is poor since the complete transitive closure of calls must be computed. If, instead of formulating the query as in Figure 2, the cl_depends was expressed using a single edge labelled with the regular expression $(contains.calls^{+}. - contains)^{+}$, the following equivalent (if applied to the same database), but more efficient, Datalog program would be produced.

```
cl_depends(class("Object"), class(Y)) :-
                      tc_conc1(class("Object"), class(Y)).
tc_conc1(X, Y) :- conc1(X, Y).
tc_conc1(X, Y) :- conc1(X, Z), tc_conc1(Z, Y).
conc1(X, Y)    :- contains(X, T), conc2(T, Y).
conc2(T0, Y)   :- tc_calls(T0, T1), contains(Y, T1).
tc_calls(X, Y) :- calls(X, Y).
tc_calls(X, Y) :- calls(X, Z), tc_calls(Z, Y).
```

As evident in Figure 2, GraphLog users are allowed to label nodes with (single) constants. Thus, any translation of such a query to Datalog will result in a program containing a single constant in the query goal, whereupon the rewriting techniques such as magic sets [1], factoring [11] and context rewriting [9] become applicable. We assume that readers are familiar with each of these.

CORAL provides an annotation for controlling evaluation called @factoring. This applies the context rewriting transformation of [9]. We will use the term *factoring* in the same way as in [11] to mean the reduction in the number of arguments in recursive predicates. Context rewriting does not do this, so to avoid confusion we will refer to the CORAL annotation as *context rewriting.*

The present translation from GraphLog to Datalog results in rules in which the only form of recursion is transitive closure (possibly with additional arguments carried in the head and recursive subgoal). The variable patterns are such that, given most binding patterns for the recursive predicate, the subgoals of the recursive rule could be reordered so that the rule is either right-linear or left-linear. As such, context rewriting is almost always applicable (if both head and tail nodes are bound to constants the resulting program might be neither left- nor right-linear). This is related to the problem of overbound queries studied in [10]. With a translation to CORAL, one could circumvent this problem by never exporting a binding pattern for a module which included both the first two arguments of a recursive predicate being bound.

In fact, the present translation always produces right-recursive rules which, depending on the binding pattern, may or may not be right-linear. Nevertheless, it would be a simple matter to direct the translation based on the bindings in the query. A more fundamental problem is that context rewriting may be an order of magnitude less efficient than a true factoring transformation.

Rather than a translation based on the structure of the original regular expression in the GraphLog query, we propose an alternative translation based on a nondeterministic finite automaton constructed from the regular expression, as originally proposed in [15]. From now on, we will refer to the former technique as the RE-*translation* and the latter as the NFA-*translation.*

The NFA-translation can result in a Datalog program containing mutually recursive rules. As a result, neither context rewriting nor the original factoring transformation in [11] are applicable. In fact, the resulting programs are also not necessarily weakly right-linear as defined by Ross in [14][2]. Nevertheless, as we demonstrate in the next section, our translation always produces a program in which every IDB predicate is factored.

3. Translating GraphLog to Factored Datalog

We assume initially that we are given a GraphLog query comprising one define query and one filter query with a single distinguished edge. Furthermore, the define query has only a single nondistinguished edge, in addition to its distinguished edge. The nondistinguished edge can be labelled with an arbitrary regular expression. The filter query simply refers to the distinguished edge of the define query.

[2]This is because the recursive subgoal may have more arguments bound than the head.

Let R denote the regular expression in the query. The first step in the translation is to construct a nondeterministic finite automaton (NFA) M which accepts the language $L(R)$. Now we essentially construct a regular grammar G from M in the standard way, followed by a regular chain program based on G. Because of space limitations, we describe only the translation when one of the nodes in the GraphLog query is labelled with a constant. If it is the head node, we first reverse the automaton M, also performing the inversion of each term labelling a transition in M.

1. Generate the fact `s(c)`, where s is the initial state of the automaton, and `c` is the node constant in the GraphLog query.

2. For each transition t from p to q labelled with $e(Z)$, where Z may be a sequence of terms, generate a Datalog rule as follows:

   ```
   q(Y) :- p(X), e(X,Y,Z).
   ```

   If, instead, t is labelled with $-e(Z)$, generate:

   ```
   q(Y) :- e(Y,X,Z), p(X).
   ```

   Call the resulting program P.

3. From P generate a new program Q by performing a bottom-up propagation of the named variables as follows.

   (a) Add each fact `s(c)` to Q.

   (b) For each rule in P containing an `s` subgoal, say,

   ```
   t(Y) :- s(X), e(X,Y,Z).
   ```

   add the rule

   ```
   t(Y,U) :- s(X), e(X,Y,Z).
   ```

   to Q, where `U` comprises all the named variables in `Z`.

   (c) Repeat the following process until no (syntactically) new rule is added to Q. If there is a rule with head `t(Y,U)` in Q and a rule of the form

   ```
   p(Y) :- t(X), e(X,Y,Z).
   ```

   in P, then add the rule

   ```
   p(Y,V) :- t(X, U), e(X,Y,Z).
   ```

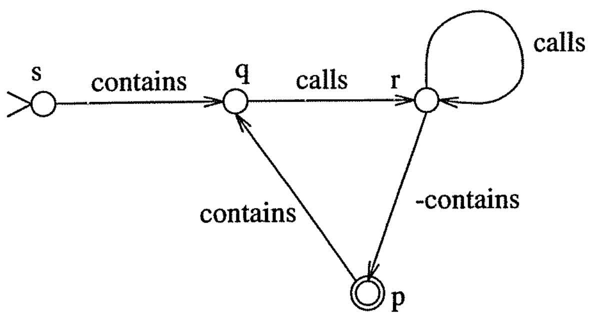


Figure 3: NFA for `(contains.calls`$^+$`.-contains)`$^+$

to Q, where `V` is a sequence of named variables which includes all of those in `Z` and `U` such that the ordering of variables is consistent throughout the set of rules.

4. Finally add rules for the distinguished edge. These rules have as their head the distinguished predicate and each body contains the predicate of greatest arity generated for each final state of the automaton.

The translation for GraphLog queries in which neither node is labelled with a constant is a simple modification of the above. Because of space limitations, we cannot describe the translation for more general queries containing more than one nondistinghuished edges, except to say that the set of nodes produced by each "edge query" can be passed to the next edge according to some traversal ordering of the query graph.

Note that, for the well-known ancestor query, the above translation effectively produces the left-linear transformation of [12] when the tail node is labelled with a constant, and produces the right-linear transformation when the head of the edge is bound to a constant.

If the GraphLog query of Figure 2 is rewritten so as to contain only one nondistinguished edge labelled with $(contains.calls^+.-contains)^+$, the associated NFA is the one depicted in Figure 3. Being at the initial state of the automaton represents being at the node of the database graph labelled `Object`. Each transition in the NFA represents the traversal of the corresponding edge. Hence, performing the transition from state s to state q represents following a `contains` edge from the node labelled `Object` to some node labelled `Y`. All such nodes are computable by the Datalog rule `q(Y) :- s(X), contains(X,Y)`. If each transition of the NFA is thus translated into a Datalog rule, evaluation of the generated program determines the set of nodes reachable from `Object` by following a path of the form $(contains.calls^+.-contains)^+$. The complete CORAL

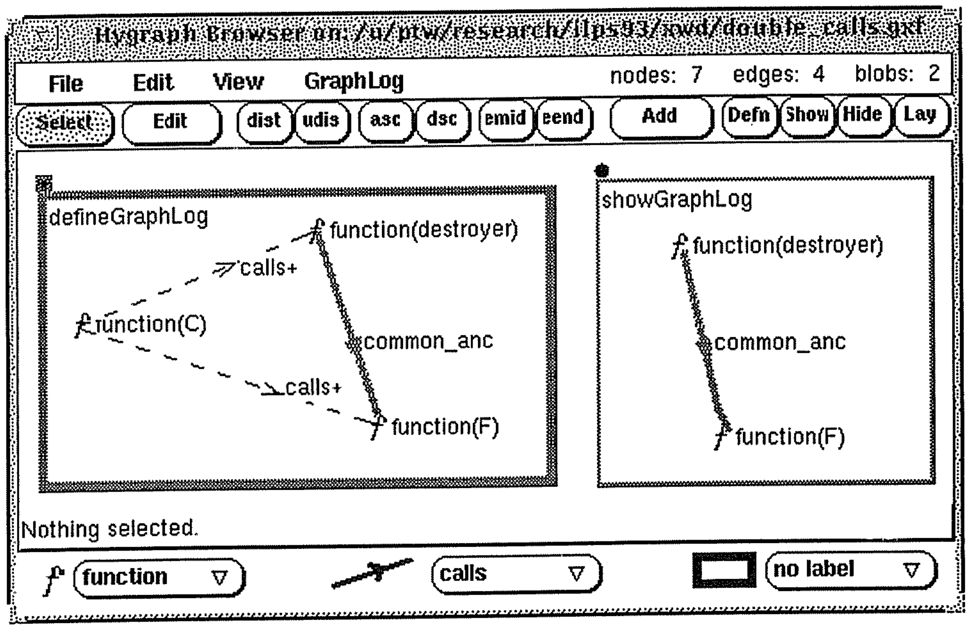


Figure 4: Another GraphLog query on the database of Figure 1.

program that is produced by the above translation is shown below.

```
s(class("Object")).
q(Y) :- s(X), contains(X, Y).
r(Y) :- q(X), calls(X, Y).
r(Y) :- r(X), calls(X, Y).
p(Y) :- contains(Y, X), r(X).
q(Y) :- p(X), contains(X, Y).
cl_depends_bf(class("Object"), class(Y)) :- p(class(Y)).
```

Note how the IDB predicates have been factored to be unary rather than binary. A performance comparison between this translation and the original is given in the next section.

Let us now consider the query shown in Figure 4, which is a form of "common ancestor" query applied to function calls. We are interested in all functions which are called (directly or indirectly) by a function which calls (directly or indirectly) the function `destroyer`.

The RE-translation yields the following:

```
common_anc(function(destroyer), function(F)) :-
                    tc_calls(function(C), function(F)),
                    tc_calls(function(C), function(destroyer)).
```

```
        tc_calls(X,Y) :- calls(X,Y).
        tc_calls(X,Y) :- calls(X,Z), tc_calls(Z,Y).
```

The subgoals in the first rule are once again poorly ordered. However, the common ancestor program is interesting because, even if the subgoals in the first rule are interchanged, it too cannot be factored using traditional techniques. The problem this time is that, no matter what ordering of subgoals is chosen, the `tc_calls` subgoal is called with two different binding patterns. This means that CORAL defaults to supplementary magic sets, with a possible order of magnitude slow down. In contrast, our factored translation is given below:

```
    s(function(destroyer)).
    anc(Y) :- calls(Y, X), s(X).
    anc(Y) :- calls(Y, X), anc(X).
    des(Y) :- anc(X), calls(X, Y).
    des(Y) :- des(X), calls(X, Y).
    common_anc_bf(function(destroyer), function(F)) :- des(function(F)).
```

Here we see that both left- and right-linear rules have been generated, once again computing only sets of nodes at each step. The performance of the two translations is compared in the next section.

4. Performance Comparisons

In this section, we present some of the performance results we obtained during our investigation. A number of different databases of facts were used in our tests; only two are reported on below. The first was a database of flight information for airlines. It comprised 112 facts for a single predicate, involving source and destination cities, as well as airline names. There were 31 distinct cities, and 6 different airlines represented.

The second database has already been mentioned, namely, the NIH C++ class library. It comprises 9124 facts, of which 2406 are `calls` facts, and 2755 are `contains` facts. Some typical results follow (all times are in seconds and are averaged over a number of runs on a lightly loaded Sun Sparc10).

			A	B	A/B	C	B/C
	query	no. of answers	RE trans.	Opt. RE trans.		NFA trans.	
1.	`alternating_bfff`	121	3.90	0.81	**4.80**	0.32	**2.52**
2.	`class_depends_bf`	32	214.94	31.31	**6.86**	1.42	**22.11**
3.	`common_anc_bf`	507	225.48	44.07	**5.12**	1.34	**32.89**

The column labelled A is the time to execute a program produced by the RE-translation with context rewriting by CORAL. Because the RE-translation does not always choose the best ordering of subgoals, we have modified the RE-translation with the best subgoal order by hand. The execution of the resulting program with context rewriting by CORAL is reflected in column B. Column C corresponds to the NFA-translation with no rewriting by CORAL.

We have already seen test cases 2 and 3: they correspond to Figures 2 and 4, respectively. Test case 1 asks for all cities reachable from Toronto using flights which alternate between a pair of airlines. Each test case was chosen to demonstrate a particular situation. Test case 1 cannot be context rewritten as produced by the RE-translation. However, if left-recursive rather than right-recursive rules are produced, context rewriting can be done. This is done by hand and is reflected in column B. Nevertheless, factoring still shows a slight improvement over context rewriting.

In test cases 2 and 3, the programs produced by the RE-translation result in complete transitive closures being computed. Execution speeds are improved by 5 to 6 times by choosing a subgoal ordering which avoids this. As a result, test case 2 is context rewritten by CORAL. However, one can show that context rewriting can be an order of magnitude less efficient than factoring on this program, a fact which seems to be borne out by the empirical results. Test case 3 as produced by the RE-translation cannot be context rewritten, no matter what subgoal ordering is chosen; hence, CORAL uses supplementary magic sets. In this case, factoring achieves a speedup of over 30.

A further indication of where context rewriting can break down is given when we want to take the *reflexive* transitive closure of the depends relationship, rather than simply the transitive closure. In the above example, the answer turns out to be the same because class `Object` depends transitively on itself. The program produced by the NFA-translation exhibits no apparent difference in execution speed. However, the program produced by the RE-translation, even with the best subgoal ordering, slows down from an execution speed of 31 seconds to one of 249 seconds.

5. Conclusion

We have discussed our experiences with the query evaluation component of the Hy$^+$ data visualization system developed at the University of Toronto. We have focussed on translating GraphLog queries into CORAL programs, providing two alternative translation strategies. In so doing, we have also demonstrated enormous differences in efficiency between equivalent Datalog programs.

The NFA-translation, in fact, also defines an alternative semantics for GraphLog which coincides with the semantics defined by the RE-translation when that produces a safe Datalog program. However, the NFA-translation can produce safe programs when the RE-translation does not. A full discussion of this is left to a future paper.

The choice between producing left- or right-linear rules when no node is labelled with a constant should also be carefully considered. We ran tests on a database representing the genealogy of theoretical computer science. When visualized as a graph, the fan-in of each node is at most two, while the fan-out can be over 20. Worst-case analysis in [10] suggests that, for such a case, left-linear rules are superior to right-linear rules. Empirical results confirmed this: right-linear rules were on average about 40% slower than left-linear rules for the ancestor program on this database.

Whether the NFA-translation has any advantage when extended to queries involving aggregation or those requiring complex filtering, which can be particularly inefficient, remains to be investigated. Also we have restricted our attention to recursive queries; however, nonrecursive queries involving large joins can also be inefficient using the present RE-translation. We hope to be able to improve on this too in the future.

References

[1] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. Sixth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 269–283, 1987.

[2] M. Consens and M. Hasan. Supporting Network Management through Declaratively Specified Data Visualizations. In H. Hegering and Y. Yemini, editors, *Proc. of the IEEE/IFIP Third Int. Symp. on Integrated Network Management, III*, pages 725–738. Elsevier North Holland, April 1993.

[3] M. Consens, M. Hasan, and A. Mendelzon. Debugging Distributed Programs by Visualizing and Querying Event Traces. In *Proc. of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 181–183, May 1993. Extended abstract.

[4] M. Consens and A. O. Mendelzon. Expressing Structural Hypertext Queries in GraphLog. In *Proc. of Second ACM Hypertext Conf.*, pages 269–292, 1989.

[5] M. Consens and A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *Proc. Ninth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 404–416, 1990.

[6] M. Consens, A. O. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *14th Int. Conf. on Software Engineering*, pages 138–156, 1992.

[7] M. Consens, A. O. Mendelzon, and D. Vista. Deductive database support for data visualization. In A. Mendelzon, editor, *Declarative database visualization: recent papers from the Hy$^+$/GraphLog project.* Technical Report CSRI-285, Univ. of Toronto, June 1993.

[8] M. Fukar. Translating GraphLog into Prolog. Master's thesis, Department of Computer Science, University of Toronto, 1991.

[9] D. Kemp, K. Ramamohanarao, and Z. Somogyi. Right-, left-, and multi-linear rule transformations that maintain context information. In *Proc. 16th Int. Conf. on Very Large Data Bases*, pages 380–391, 1990.

[10] I. S. Mumick and H. Pirahesh. Overbound and right-linear queries. In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 127–141, 1991.

[11] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Argument reduction by factoring. In *Proc. 15th Int. Conf. on Very Large Data Bases*, pages 173–182, 1989.

[12] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and combined-linear rules. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 235–242, 1989.

[13] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proc. 18th Int. Conf. on Very Large Data Bases*, 1992.

[14] K. A. Ross. Modular acyclicity and tail recursion in logic programs. In *Proc. Tenth ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Databases Systems*, pages 92–101, 1991.

[15] P. T. Wood. Factoring augmented regular chain programs. In *Proc. 16th Int. Conf. on Very Large Data Bases*, pages 255–263, 1990.

IMPROVING DATA QUALITY VIA LDL++

Shawn Shaw, Lena Foggiato-Bish, Iris Garcia, Gherin Tillman,
Dave Tryon, Wendy Wood,Pacific Bell
Carlo Zaniolo, University of California, Los Angeles

INTRODUCTION: THE BUSINESS CHALLENGE

The MIS functions of modern enterprises must cope with rapidly increasing demands and requirements. Even as the complexity and interconnection of applications increase, the ever increasing competitive marketplace their companies are now existing in is exercising a downward pressure on the time available for the development lifecycle. These same competitive pressures, however, are increasing the importance of multiple application inter-operability and semantic integration.

In this paper, we elucidate the reasons for which the deductive database technology can play an important role in answering this business challenge, and then present the case-study of two "data cleaning" applications under development at Pacific Bell. These case-studies confirm the potential of deductive databases but also illustrate the practical challenges, such as interoperability with other programming environments and DB2 databases, that one faces in establishing this new technology in the business world. This paper is not intended to demonstrate the legitimacy of LDL and LDL++ as deductive database technology; this has been discussed elsewhere [Ceta, KNZ, NaTs, Ts1, Ts2, Ullm, UlZa]. We are concerned with LDL++'s ability to interconnect and accomplish meaningful, practical work.

THE ROLE OF DATABASE TECHNOLOGY IN MEETING THE CHALLENGE

In addition to the traditional functions of database technology: shareability, persistency, concurrence control, integrity and recovery; modern database technology is being asked to directly enforce constraints contained in the users ' requirement and specifications. Specifically, in addition to the traditional expectations of database technology, such as integrity, sharing and recovery, these new applications demand the consistent enforcement of complex structures and recursively defined objects through the use of high-level languages and rules. Conventional database systems fail to address the special requirements of such applications.

Conventional database systems also suffer from the limited power of their query languages. Since conventional query languages, as exemplified by SQL, are capable of accessing and modifying data in only limited ways, database applications are now written in a conventional, procedural language intermixed with query language calls. Since the non-procedural, set-oriented computational model of SQL is so different from that of procedural languages, and because of incompatible data types, an "impedance mismatch" occurs that hinders application development and causes expensive run-time conversions.

THE NEED FOR DEDUCTIVE DATABASE TECHNOLOGY

For these reasons, and others, modern applications need a single, computationally complete language that answers the needs previously discussed serving both as a query language and as a general-purpose host language.

Relational languages are declarative and logic-based, but as implemented in commercial relational database products, not supportive of a full logic, and have, therefore, been limited. Nevertheless, despite the historical limitations, declarative languages are the right direction. They provide the ability to express what one wants, and leaves to the system substantial portions of the algorithm required to meet the request. This ability is essential for ease of use, data independence and code reusability.

Deductive databases represent relational languages which are both declarative and supportive of a full logic. They take the declarative approach providing a declarative, logic-based language for expressing queries, reasoning, and complex applications on databases. But note that deductive databases are database technology, not merely a programming language.

DEDUCTIVE DATABASES: MORE THAN A PERSISTENT PROLOG

A few words about Prolog. A new generation of powerful rule-based languages for expert systems applications commanded great attention in the 80's. Among these, Prolog is of particular interest, which has led to considerable work at building a deductive database system, either, by coupling Prolog with relational DBMSs or by extending Prolog with database capabilities [CGT], which retained Horn clauses with their rule-oriented syntax and aiming to achieve a complete and harmonious integration of logic and databases retained Horn clauses with their rule-oriented syntax. It produced new languages and systems that combine the database functionality and non-procedurality of relational systems, with Prolog's reasoning and symbolic manipulation capability. Such systems have been further enriched with knowledge representation and object-oriented constructs. Using extensions of relational DBMSs technology, a new implementation technology was developed for these languages to ensure their efficient support on, both, main memory and secondary store. Among those systems [Meta,KiMS], LDL++ has reached the most maturity [Ceta,ArOn]. Fully integrated deductive database systems have the following distinguished traits:

Unlike procedural languages, Deductive Database technology is oriented toward a specification-based and declarative style of computing. Indeed, DBMSs make extensive use of data dictionaries, and rely on multiple layers of data definition and specifications (storage schema, schema, sub-schemas, views, etc). Furthermore, like relational DBMSs, the Logic Database system supports a high-level, logic-based language whereby users need only to specify high-level queries, and can leave the responsibility for making and implementing performance-oriented decisions to the system. Significant gains in programmer productivity, data independence, and maintenance (i.e., over the whole software life cycle) follow from this approach.

Frequently, in database design problems arising from incompleteness or inconsistencies in the users' requirement are not discovered until after one has progressed deep into the development lifecycle . Furthermore, discrepancies between the users' requirements and expectations and the actual implementation do not emerge until the first running implementation is completed (after great expenditures of time and resources).

Logic-based rule systems, such as Prolog, enable the rapid prototyping of complex applications beyond what is possible using other systems. Their clear logic-based semantics enhance the value of the resulting prototype as an important specification and validation artifact in the development cycle. These particular merits of logic-based rule systems for specification driven re-engineering are extended by deductive databases, with their strong relational database orientation and their emphasis on declarative, formal semantics. Indeed, deductive databases can be defined, in a nutshell, as the unification of logic-based rule systems and database technology.

LDL++: A DEDUCTIVE DATABASE TECHNOLOGY

Originally developed at MCC, LDL (and now its successor LDL++) is one of the first actual examples of deductive database technology. In addition to the traditional database requirements, such as integrity, sharing and recovery, LDL++ supports complex structures, recursively defined objects, set operations, update and retrieval through query forms, and inter-operability with high and low level languages, and remote access to other database management systems. In addition, LDL++ incorporates several useful concepts of object-oriented technology including object-identity and a rich type structure with inheritance of properties from types to their subtypes. However, in establishing object identity, LDL++ retains the "value-orientation" of relational systems, the declarative, logic-based essence, and does not become "tainted" with "proceduralism" as is Prolog. Thus, in summary, LDL++ supports ease of use, data independence and code reusability through a declarative, logic-based language for both the retention of its extensional database facts and for the expression of queries, reasoning, and complex operations.

In particular LDL++ supports:

* The notion of recovery and database transactions. These concepts are deeply ingrained in the semantics of the LDL, where an error condition, such as the isolation of an integrity constraint, will result in the undoing of all the updates after the last checkpoint.

* Unique key constraints.

* Non-linear rules or cyclic graphs. These are much simpler to write in LDL than in Prolog or a procedural language.

* Query optimization. The notion of a query optimizer is also part of LDL++ system [Ullm,Ceta,ArOn], increasing compatibility with relational systems, better data independence, and enhanced program reusability.

* Stratified negation, Grouping and Non-deterministic pruning extending LDL++ beyond simple Horn Clauses [NaTs].

A FRAMEWORK FOR THE PRACTICAL APPLICATION OF LDL++ IN THE BUSINESS ENVIRONMENT: DATA QUALITY RE-ENGINEERING

It is always a challenge to establish a new technology in the business environment. The shift to an untried, unknown technology is seen (correctly) as a significant business risk. Until deductive database technology is incorporated into the DBMS product line of major database vendors, large companies will shy away from utilizing it in any major application area which is either mission critical to the corporation or which must be stable over a significant period of time (i.e., years, decades, etc.). Therefore, first applications must be in support of business functions which already represent areas of difficulty and chaos to the company and which have a short (preferably one-time) payback cycle.

For these reasons we are introducing deductive technology into commercial use in the area of data quality, that is, the "scrubbing" of "dirty data" from our suite of corporate databases. This activity is beginning to be known under the term Data Quality Re-engineering. Data Quality Re-engineering represents a near perfect environment for "trying out" this technology. The databases are already a mess, and the existing applications are incapable of resolving the issues. If they were capable, they would have done it! Also the use of the technology adds value with each instance of usage. Thus, data quality is a reasonable arena for the first commercial use.

But, before we can utilize the technology to "scrub" the databases of existing applications, we have to be able to communicate with those applications existing data stores. That is where the data "lives."

Therefore, the first capability to be demonstrated is the inter-operability of LDL++ with other languages and database environments. Then we must show that in this heterogeneous data world, LDL++ can, in fact, be used to find and identify "dirty data" in a commercial database.

PROGRAM 1: LDL++ INTER-OPERABILITY WITH C++

Business Context

In the course of provisioning service to a telecommunications customer, a circuit is associated with a network interface. Due to our multiple legacy application systems environment, records of this relationship are kept in multiple, independent databases. The business problem is verifying that these records are consistent within these multiple systems. For the purpose of this demonstration, we are focusing on the record of circuit to network interface found in a customer contact application (CESAR), the facilities tracking system (FS), and the inter-exchange carrier billing system (CABS). A failure to maintain global logical consistency among these multiple applications could result in engineering being unable to complete their work forcing an extraneous customer contact, the billing system to produce inaccurate bills, or the customer not to receive their requested services.

The data representing the relationship between circuit and network interface was extracted from each of these application systems, joined on circuit identifier, and presented as a "flat file" to LDL++ in the UNIX environment. This pre-joining of the data will not be necessary in the future with the coming availability of a concurrent DB2 accessing capability . Single DB2 remote access will be demonstrated in the second demo program.

Technical Context

The purpose of demonstration program 1 is to show the ability of LDL++ to call a C++ object to perform a specified substring manipulation, a feature not currently supported within LDL++ itself. Once established, this capability can be used in other situations to allow the extensive use by LDL++ of predefined C++ object libraries. Conversely, LDL++ can be called by C++ opening the door on the use of LDL++ to provide a formally robust database environment to C++ applications. We see this as the first small step into a future in which deductive database engines will provide a logically sound, industrial grade database repository underlying object-oriented, user-facing applications.

Job Run Procedure

Step 1. The client uses SQL and/or special programs to fetch the data from three databases, join them together, and load them into the UNIX environment.

Step 2. Using AWK or C++, convert the data to LDL++ database predicate format. as in the following example:

```
p('ALN', 1, 'CIRCUITID01X-001', 'XH--', '04DS9.15', '04DU5.56', 'YH--', '04DS9.15',
        '04DU5.56', 'XH--', '04DS9', '04DU5', '04DS9', '04DU5', '04DS9', '04DU5').
```

Step 3. Code the LDL++ program and C++ module. The following is an example extracted from a recent successful data quality project in which LDL++ played a significant role [GARC]. The full LDL++ program contained more than 10 pages of complex rules and took several hours for the live production run.

LDL++ PROGRAM

```
database( { p(string,integer,string,string,string,string,string,string,
          string,string,string,string) }).

import foreign
ldl_substring($Str: string, $Begin: integer, $Size: integer, SubStr: string) from
'ldl_substring.o'.

%          ------  QUERY FORM  ------
export duplicate_rcds(X,B).
export mismatch_NCC(X,B,C,F,I).
export mismatch_NCI(X,B,D,G).
export mismatch_NCI2(X,B,E,H).
export mismatch_FS_NCI(X,B,J,L,N).
export mismatch_FS_NCI2(X,B,K,M,O).

export rule_broken(X,A,B,C,D,E,F,G,H,I,J,K).

% RULE 1.
% none of the circuit id's should be the same

duplicate_rcds(X,B) <-
        p(_,X,B,_,_,_,_,_,_,_,_,_), p(_,X1,B,_,_,_,_,_,_,_,_,_),
     X ~= X1.

% RULE 2.
% FS's NCC should be same as CEASAR's & CABS's NCC

mismatch_NCC(X,B,C,F,I) <-
        p(A,X,B,C,D,E,F,G,H,I,J,K),(C,C) ~= (F,I).

% RULE 3.
% CEASAR's NCI should be same as CABS's NCI

mismatch_NCI(X,B,D,G) <-
        p(A,X,B,C,D,E,F,G,H,I,J,K),D ~= G.

% RULE 4.
% CEASAR's NCI2 should be same as CABS's NCI2
```

```
mismatch_NCI2(X,B,E,H) <-
        p(A,X,B,C,D,E,F,G,H,I,J,K),E ~= H.

% RULE 5.
% (first 5 chara of) FS NCI should be same as first 5 chara of CEASAR & CABS'
NCI.

mismatch_FS_NCI(X,B,J,L,N) <-
        p(A,X,B,C,D,E,F,G,H,I,J,K),
      ldl_substring( D, 1, 5, L),
      ldl_substring( G, 1, 5, N),
      (J,J) ~= (L,N).

% RULE 6.
% (first 5 chara of) FS NCI2 should be same as first 5 chara of CEASAR &
CABS' NCI2.

mismatch_FS_NCI2(X,B,K,M,O) <-
        p(A,X,B,C,D,E,F,G,H,I,J,K),
      ldl_substring( E, 1, 5, M),
      ldl_substring( H, 1, 5, O),
      (K,K) ~= (M,O).

% Summarize RULE 2 TO RULE 6 into one single LDL++ rule.

rule_broken(X,A,B,C,D,E,F,G,H,I,J,K) <-
        p(A,X,B,C,D,E,F,G,H,I,J,K),
      ldl_substring( D, 1, 5, L),
      ldl_substring( G, 1, 5, N),
      ldl_substring( E, 1, 5, M),
      ldl_substring( H, 1, 5, O),
        (C,C,D,E,J,J,K,K) ~= (F,I,G,H,L,N,M,O).


/***********************************************************
 *          This is C++ module called by LDL++.
 * This function creates a sub-string given the starting location
 * and the size -
 ***********************************************************/

#include <string.h>
#include <stream.h>
#include <ldl_extern.h>

// Make sure that the name is not mangled
extern "C"
```

```
{
 LdlStatus ldl_substring(LdlObject str, LdlObject begin, LdlObject size, LdlObject
substr); }

LdlStatus ldl_substring(LdlObject str, LdlObject begin, LdlObject size, LdlObject
substr)
{
 LdlStatus    status = LDL_FAIL;

 if (ldl_entry_p())
   {
    if ((ldl_string_p(str))
       && (ldl_int_p(begin))
       && (ldl_int_p(size)))
      {
       char*        str_value;
       int          begin_value;
       int          size_value;

       if ((str_value = ldl_get_string(str))
          && (begin_value = ldl_get_int(begin))
          && (size_value = ldl_get_int(size)))
         {
          int      str_len;

          str_len = strlen(str_value);

          if ((str_len > 0)
             && (begin_value > 0)
             && (size_value > 0)
             && (begin_value <= str_len))
            {
             LdlObject    local_substr;

             if (begin_value + size_value - 1 < str_len)
               {
                char     temp_ch;
                int      null_location;

                null_location = begin_value + size_value - 1;
                temp_ch = str_value[null_location];
                 str_value[null_location] = '\0';
                local_substr = ldl_create_string(&str_value[begin_value-1]);
                str_value[null_location] = temp_ch;
               }
             else
              local_substr = ldl_create_string(&str_value[begin_value-1]);
            ldl_assign_argument(substr, local_substr);
            ldl_delete_object(local_substr);
            status = LDL_SUCCESS;
           }
         else
           cerr << "\nExternal Error (substring): Invalid arg values" << endl;
        }
      else
        cerr << "\nExternal Error (substring): Can not get arg values" << endl;
     }
   else
     cerr << "\nExternal Error (substring): Illegal arg types" << endl;
  }

 return(status);
}
```

Step 4. Run the LDL++ program:

```
ldl++(1)> open ../iris2.rul

Opening file : ../iris2.rul ... done.

ldl++(2)> compile

Compiling all query forms in global module ... done.

ldl++(3)> initdb moreTst.fac

Initializing database from moreTst.fac ... done.

ldl++(4)> query duplicate_rcds(X,B)

Querying : duplicate_rcds(X,B) ...
    (1, CIRCUITID01X-001)
    (6, CIRCUITID06X-023)
    (7, CIRCUITID07X-023)

    The number of records is 3.

ldl++(5)> query mismatch_NCC(X,B,C,F,I)

Querying : mismatch_NCC(X,B,C,F,I) ...
    (1, CIRCUITID01X-001, XH--, YH--, XH--)
    (2, CIRCUITID02X-006, XH--, XH--, ZH--)
    The number of records is 2.

ldl++(6)>  query  mismatch_NCI(X,B,D,G)

Querying : mismatch_NCI(X,B,D,G) ...
      (3, CIRCUITID03X-023, 04DS9.15, 04DS9.33)

    The number of records is 1.

ldl++(7)>  query  mismatch_NCI2(X,B,E,H)

Querying : mismatch_NCI2(X,B,E,H) ...
      (4, CIRCUITID04X-021, 04DU5.56, 04DU5.44)

    The number of records is 1.

ldl++(8)>  query  mismatch_FS_NCI(X,B,J,L,N)

Querying : mismatch_FS_NCI(X,B,J,L,N) ...
      (5, CIRCUITID05X-022, 55555, 04DS9, 04DS9)

    The number of records is 1.

ldl++(9)>  query  mismatch_FS_NCI2(X,B,K,M,O)

Querying : mismatch_FS_NCI2(X,B,K,M,O) ...
      (6, CIRCUITID06X-023, 66666, 04DU5, 04DU5)

    The number of records is 1.
```

LDL++ INTEROPERABILITY WITH ANOTHER DATABASE: LDL++ AND DB2

Business Context

Many (most) applications within our corporation use the Common Location Edit (CLEDIT) database application to validate location information associated with company equipment, premises, and service. Within the various types of location information, a large number of multi-valued dependencies exist. The existence of invalid combinations of location information within the CLEDIT application allow invalid patterns of information to become established within multiple independent applications databases. The business problem is to prevent invalid combinations from becoming established within the CLEDIT reference tables and also provide the capability to access independent applications and "scrub" their existing databases for "old garbage" which may have been established in the past.

Technical Context

The purpose of demonstration program 2 is to show the ability of LDL++ to access a remote DB2 database. The particular cardinality constraint being checked is that the combination of geographic code, parcel code, and property index maps to a single location code.

LDL++ PROGRAM

```
----------------------------------------------------------------------------
%
% Demo for showing the integration of LDL++ with DB2
% using the NDB server.
%

database( {     db2::'fsys.cledt_cloc'( cledt_loc_id:CHAR(16),
                         loc_pactel_cd:CHAR(11),
                         geo_terr_cd:CHAR(2))
                local_name      cledt_cloc
                from            netdbsrv
                use             fsys
                user_name       sxtsou
                host_name       techlab
                 client_ip_address '129.34.223.10'
                rpc_program_number 536870992
                rpc_version_number 1
                password        xxxx,

            db2::'fsys.cledt_parc'( cledt_loc_id:CHAR(16),
                         actg_parc_cd:CHAR(3),
                         prop_idx_cd:CHAR(1))
                local_name      cledt_parc
                from            netdbsrv
                use             fsys
                user_name       sxtsou
                host_name       techlab
                 client_ip_address '129.34.223.10'
                rpc_program_number 536870992
                rpc_version_number 1
                password        xxxx}).

% Query forms

export answer(W,Z).
export answer($W,Z).
export all($A,B,C,D).
export group($A,B).
export duplicate($A,B).
```

```
% Rules

answer(W,Set) <- duplicate(W,U), group(W,Set).
duplicate(LocPactelCd,<(GeoTerrCd1, ActgParcCd1, PropIdxCd1)>) <-
                    all(LocPactelCd, GeoTerrCd1, ActgParcCd1, PropIdxCd1),
                    all(LocPactelCd, GeoTerrCd2, ActgParcCd2, PropIdxCd2),
                    (GeoTerrCd1, ActgParcCd1, PropIdxCd1) ~=
                                (GeoTerrCd2, ActgParcCd2, PropIdxCd2).

group(LocPactelCd,<(GeoTerrCd, ActgParcCd, PropIdxCd)>) <-
      all(LocPactelCd, GeoTerrCd, ActgParcCd, PropIdxCd).

all(LocPactelCd, GeoTerrCd, ActgParcCd, PropIdxCd) <-
            cledt_cloc(CledtLocId, LocPactelCd, GeoTerrCd),
            cledt_parc(CledtLocId, ActgParcCd, PropIdxCd).

/*    log

ldl++(6)>  query  all('AVLNCA11',B,C,D)

Querying : all('AVLNCA11',B,C,D) ...

      (AVLNCA11, AD, 00A, B)
      (AVLNCA11, AD, 100, A)
      (AVLNCA11, AD, 100, B)

      The number of records is 3.

ldl++(9)>  query  answer('AVLNCA11',B)

Querying : answer('AVLNCA11',B) ...

      (AVLNCA11, {(AD, 00A, B), (AD, 100, A), (AD, 100, B)})

      The number of records is 1.
```

CONCLUSION

In this paper we have shown the ability of a deductive database (LDL++) to support inter-operability with other programming languages and databases. In addition, we have used this interconnection to perform several simple searches for "dirty data."

Now that connection is established, we intend to continue investigating the capabilities of the declarative, deductive database environment to uncover, correct, and prevent the corruption of large, independent applications. Of particular interest is the potential of LDL++ to serve as the underlying database engine for C++ development.

It is only through the deployment of new, productive, and adaptable technology, such as deductive database, that the MIS functions of modern enterprises will be able to effectively respond to the rapidly increasing demands and requirements placed upon them by modern business. . The ever increasing competitiveness of the marketplace will continue to increase the importance of multiple application inter-operability and semantic integration.

References

[ARON] Natraj Arni and KayLiang Ong, "The LDL++ User's Guide,"Edition 2.0, MCC Technical Report Carnot-012-93(P), January 1993.

[Ceta] Chimenti, D. et al., `"The *LDL* System Prototype," *IEEE Journal on Data and Knowledge Engineering*, March 1990.

[CGT] Ceri, S., G. Gottlob and L. Tanca, *Logic Programming and Deductive Databases*, Springer-Verlag, 1989.

[DM89] "The Rapid Prototyping Conundrum", DATAMATION, June 1989.

[Gane] Gane, C.; *Rapid System Development*, Prentice Hall, 1989.

[GARC] Iris Garcia, NC/NCI Dta Quality Project, Pacific Bell, 1993.

[Hopp] Hopper, D.E., `"Rattling SABRE---New Ways to Compete on Information," Harvard Business Review, May-June 1990, pp. 118-125.

[KiMS] Kiernan, G., C. de Maindreville, and E. Simon; "Making Deductive Database a Practical Technology: a step forward;" *Proc. 1990 ACM--SIGMOD Conference on Management of Data*, pp. 237-246.

[KNZ] Krishnamurthy, S. Naqvi and Zaniolo, "Database Transactions in *LDL*", *Proc. Logic Programming North American Conference*, pp. 795-830, MIT Press, 1989.

[KuYo] Kunifji S., H. Yokota, "Prolog and Relational Databases for 5th Generation Computer Systems," in *Advances in Logic and Databases, Vol. 2*, (Gallaire, Minker and Nicolas eds.), Plenum, New York, 1984.

[Meta] Morris, K. et al. "YAWN! (Yet Another Window on NAIL!), *Data Engineering*, Vol.10, No. 4, pp. 28--44, Dec. 1987.

[Moss] Moss, C., "Cut and Paste---defining the Impure Primitives of Prolog", *Proc. Third Int. Conference on Logic Programming*, London, July 1986, pp. 686-694.

[NaTs] Naqvi, S. A., S. Tsur; *A Logical Language for Data and Knowledge Bases*, W. H. Freeman, 1989.

[Rama] Ramakrishnnan, R., D. Srivastava and S. Sudarshan,"CORAL--Control, Relations and Logic," Proceedings of the 18th VLDB Conference, 1992.

[Ts1] Tsur S., "Applications of Deductive Database Systems," *Proc. IEEE COMCON Spring '90 Conf.*, San Francisco, Feb 26-March 2.

[Ts2]Tsur S., "Data Dredging," *Data Engineering*, Vol. 13, No. 4, IEEE Computer Society, Dec. 90.

[Ullm] Ullman, J.D., *Database and Knowledge-Based Systems*, Vols. I and II, Computer Science Press, Rockville, Md.,1989.

[UlZa] Ullman, J. and C. Zaniolo, "Deductive Databases, Achievements and Future Directions," *SIGMOD Record*, pp. 77-83, Vol. 19, No. 4, ACM Press, Dec. 1990.

Using $\mathcal{LDL}$++ for Spatio-Temporal Reasoning in Atmospheric Science Databases [1]

Richard R. Muntz, Eddie C. Shek
and
Carlo Zaniolo

Computer Science Department
University of California
Los Angeles, CA 90024
{muntz, shek, zaniolo}@cs.ucla.edu

Abstract

We describe a system being built at UCLA to support spatio-temporal analysis and queries on very large databases of atmospheric data. Deductive database technology is being used for detecting and tracking evolving physical phenomena on massive data sets produced by combined atmospheric and ocean global models.

The architecture of our system is based on the tight coupling of an $\mathcal{LDL}$++ system with external database systems, such as Postgres and Quilt. This provides a computing environment where complex queries and reasoning on spatial and temporal data can be easily expressed and efficiently supported.

1 Introduction

Modern information systems are faced with the challenge of supporting increasingly sophisticated applications spanning several databases and a spectrum of

[1]Work partially supported by Hughes Aircraft, under the University of California MICRO-92 program, and by NASA under the HPCC/ESS NRA-92 program .

data types ranging from traditional alphanumeric data to multimedia information. Many of these applications require complex analysis to be performed in the spatial and temporal dimensions to discover important phenomena and monitor trends taking place in the application domain under investigation. Applications range from earth and space sciences to resource exploration, selective marketing and sales analysis.

The objective of this research project is to provide new technology whereby complex queries and reasoning on temporal and spatial data can be (i) easily expressed, (ii) efficiently supported and (iii) integrated with similar technology on traditional data. Toward the goal, we are

- developing an integrated model for spatial and temporal data and efficient representation and reasoning on spatial-temporal knowledge.
- building a prototype to serve as a demonstration vehicle for the proposed technology, and as a testbed for measuring the effectiveness and performance of the proposed solutions.

Our approach consists of building primitives for expressing and modeling spatial and temporal relationships and patterns on top of a deductive database system, that currently supports rule-based reasoning, and the rapid prototyping of intelligent database applications. Our objective is to construct a powerful system which will facilitate the task of describing, searching for, and making complex decisions on multimodal data patterns (spatial, temporal and alphanumeric).

This paper is organized as follows. In section 2 we describe the requirements of the application domain and our pilot implementation. The design of a language that integrates $\mathcal{LDL}$++ and EPL and which supports spatial queries is discussed in section 3. The architecture is described in section 4 and section 5 concludes the paper.

2 Application

We are working with atmospheric scientists on the analysis of atmospheric data. We will first use a synthetic data set generated from the UCLA coupled general circulation model AGCM [1, 7] which combines atmospheric and ocean global models. The simulation data is free of incomplete or contradicting information, and thus it will facilitate the validation of our prototype environment. This will then be applied and enhanced to deal with measurement data and satellite images.

AGCM produces values on horizontal velocity, potential temperature, water vapor and ozone mixing ratio, surface pressure and ground temperature. It also stores fields corresponding to diagnostic variables, such as precipitation, cloudiness, surface fluxes of sensible and latent heat, surface wind stress and radiative heating on each grid on the surface of the earth. Typically, output is written out to the database at 12-hour (simulation time) intervals, but this can be controlled so the model can be run with different resolutions. The lowest resolution version of the coupled GCM produces approximately 5 Gbytes of data per simulated year. A 100-year long GCM simulation with high-resolution generates approximately 50 Tbytes of output.

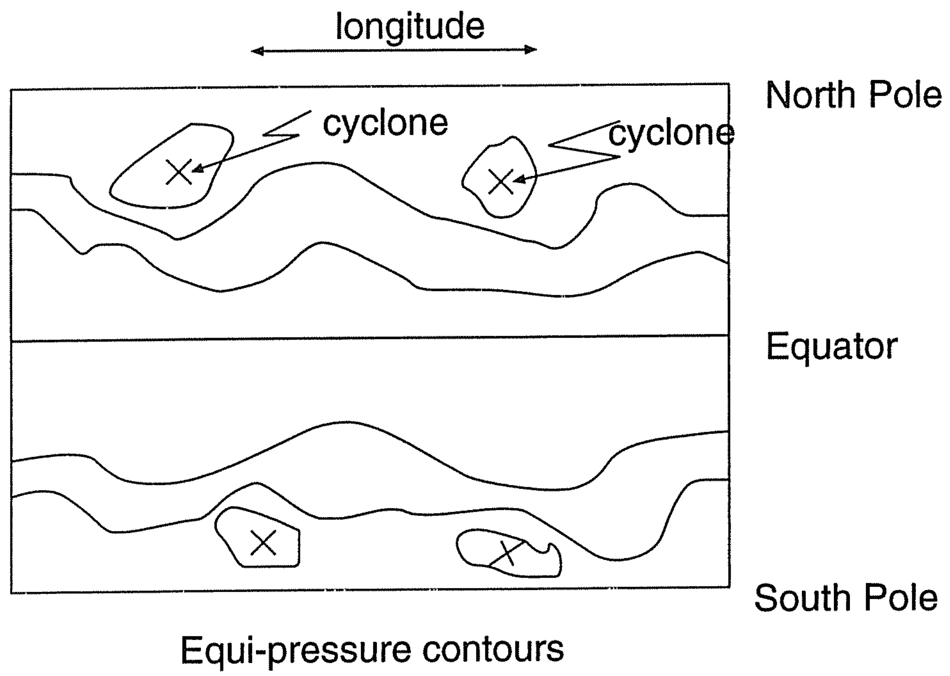


Figure 1: A Cyclone Map

We are interested in capturing features (usually spatial) and tracking them over time, one example is the monitoring of cyclone tracks and cloud formations. We define a cyclone as an area in which circular geostatic winds are occurring. As illustrated in Figure 1, the center of a cyclone is a local mini-

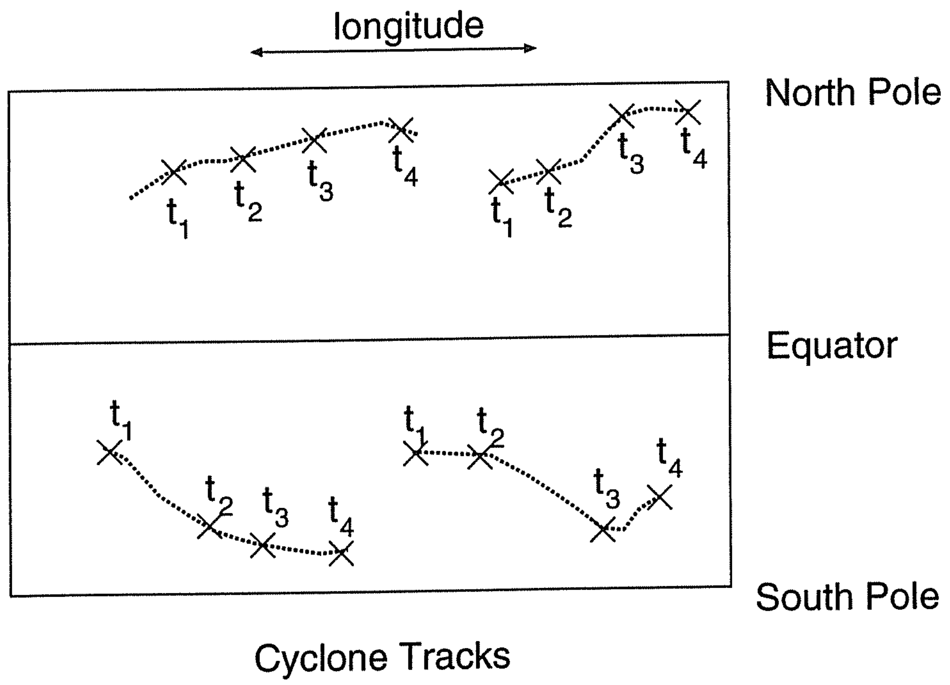


Figure 2: Cyclone Tracks at Sea Level Over Time

mum in the sea level pressure and its extent is the largest isobar that forms a closed contour enclosing the local minimum. The sea level pressure at grid points over the earth are provided by the AGCM at regular time intervals. At successive time intervals, a cyclone appears at different points, and a sequence of such points for a given cyclone is called a "track" or trajectory (see Figure 2). In addition, the maturity of a cyclone at a certain time is closely related to the shape of its associated cloud formation. So cloud formation type becomes another time-varying attribute of a cyclone.

3 The Approach

Deductive databases [8, 4, 9, 10] extends relational database technology by supporting complex objects and queries, rule-base reasoning, and rapid prototyping through the use of a logic-based language. Although very little work has been done to integrate spatial and temporal information in a deductive database environment up to now[14, 12], we believe that a logic-based deductive database language can allow scientists to conveniently pose complex queries and reason about spatial and temporal knowledge.

3.1 $\mathcal{LDL}$++

$\mathcal{LDL}$++[18] is a second-generation deductive database system. It is based on Horn clause logic, a subset of first-order predicate calculus, which gives it superior expressiveness over traditional database query languages like SQL or QUEL. To enhance the usefulness of $\mathcal{LDL}$++ in GIS applications, we are extending $\mathcal{LDL}$++ with spatial and temporal constructs. These new basic constructs can be combined freely to represent complex spatio-temporal knowledge and allow reasoning on that knowledge.

3.1.1 Temporal Extensions

Traditional DBMSs only allow queries on the current state of the database. However, our application involves tracking of phenomena in time, which is very common in GIS applications. As a result, we need a mechanism that allows us to easily and efficiently reference the state of an object at different times.

As defined in [13], the transaction time of a fact in temporal databases refers to the time when the fact is stored in the database, and from that time it is reflected in the results of subsequent queries. It may be sufficient for traditional applications, but for atmospheric science applications, data collection and processing is time-consuming, and hence the updating of the database with the collected data and any derived information will be delayed. Therefore, the database must support valid time, which is the time when the fact is true in a modeled reality. Temporal selection of objects based on valid time can be supported in $\mathcal{LDL}$++. Temporal selection, similar to spatial selection, is analogous to selection in relational algebra. Each predicate and object is augmented with a timestamp which marks their valid times. $X[t]$ denotes an object X's state at time t. The predicate $pred[t](X, Y)$ means the relationship *pred* between objects X and Y is true at time t, and remains so until changed at a later time $t' > t$; it is equivalent to $pred(X[t], Y[t])$. Note that the time annotation only makes sense for temporal objects (whose state is time varying); the annotation doesn't change the value of constants. For example, $coord(10, 10)[t]$ is equivalent to $coord(10, 10)$ for all times t since the coordinate $(10, 10)$ represents the same location regardless of time.

We are also investigating extensions that make it easier to express complex event patterns in a rule-based form. Therefore, we have designed and implemented an event pattern language (EPL) on top of $\mathcal{LDL}$++. This allows the specification of complex event patterns as part of the query goals. For example, the following rule:

```
formation_time(X) <- ep(+cyclone(Y)), X = Y.evtime.
```

can be used to get the time of formation of cyclones. The EPL predicate `ep(<event-pattern>)` is true when there is a sequence of events that matches `<event-pattern>`.

An event pattern is expressed as a regular expression of primitive events augmented with predicates. Primitive events in EPL are update events to a database table or a clock event. Update events of a database table are of the following forms:

- `+<table-name>(X)` : insertion of a tuple `X` into `<table-name>`.
- `-+<table-name>(X)` : update of a tuple `X` in `<table-name>`.
- `-<table-name>(X)` : deletion of a tuple `X` in `<table-name>`.

Given events e_1, ..., e_n, and predicates p_1, ..., p_n, the following event pattern constructors:

- *any*: any primitive event,
- $(\sim e_1)$: an event pattern that does not match e_1,
- $\wedge(*, e_1)$: a sequence of zero or more e_1's,
- $\wedge(n, e_1)$: a sequence of n e_1's,
- $< e_1, e_2, ..., e_n >$: a sequence consisting of e_1, immediately followed by e_2, ..., immediately followed by e_n,
- $[e_1, e_2, ..., e_n]$: $< e_1, \wedge(*, any), e_2, \wedge(*, any), \ldots, \wedge(*, any), e_n >$,
- $\{e_1, e_2, ..., e_n\}$: an event pattern that matches any of e_i $(1 \leq i \leq n)$,
- $e_1, p_1, ..., p_n$ = an event pattern e_1 satisfying $p_1, ..., p_n$,

3.1.2 Spatial Extensions

We also propose to extend $\mathcal{LDL}$++ to support spatial reasoning by introducing the basic 2-D spatial data types *point*, *line*, *polygon*, and *rectangle*. The logical representation of a spatial object can be derived from its physical representation. Different geometric objects have different set of geometric properties. A geometric object can be logically represented as a set of geometric features.

$$geom \subset \{location, extent, length, direction, slope, size, shape\}$$

For example, a point being a 1-dimensional geometric object, only has a location and empty extent. $point = \{location, extent\}$. Besides selecting facts based on the values of their alpha-numeric attributes as in traditional database applications, we also want to be able to retrieve spatial data based on their spatial properties.

A set of predicates are defined on the spatial data types. They are efficiently supported by hierarchical spatial data structures (e.g. quadtree[11]), and can be categorized as spatial selection and spatial join operators.

A spatial select operator selects spatial objects on the basis of their geometric properties. Spatial selection is analogous to relational select. It either retrieves a geometric property (size, location,etc) of a geometric object or identifies facts that satisfy some predicate on the spatial properties of an object.

- area(\$X, Area)[2]: evaluates the area of polygon X.
- perimeter(\$X, Perimeter): evaluates the perimeter of polygon X.
- centroid(\$X, Centroid): finds the location of polygon X's centroid.

On the other hand, spatial join predicates join spatial objects based on their spatial relationship with some other spatial objects. They evaluate and identify interrelationships between sets of spatial objects.

- in_window(X, \$Window): finds all spatial objects Xs that lie in a rectangular window.

[2]In the declaration of query forms in $\mathcal{LDL}$++, bound arguments are prefixed with an \$. The other arguments are free. Our system also supports query forms with bounded arguments in place of free arguments.

- closest_to(X, $Point): finds the spatial object that is closest to a point.
- within(X, $Y, $Distance): finds all spatial objects Xs that lie within certain distance from spatial object Y
- at(X, $Point): identifies all spatial objects Xs at a point.
- passthrough(X, Y): identifies all lines Xs that passes through polygons Ys.
- adjacent(X, Y): identifies pairs of spatial objects that lie adjacent to each another.
- contain(X, Y): identifies all spatial objects Xs that is contained in Ys. A polygon can contain points, lines, and polygons, while a line can only contain points or lines.
- intersect(X, Y, Z): locates the intersection Z of spatial objects X and Y, which can either be a line or polygon. The intersection of 2 polygons is a polygon, the intersection of a line and a polygon is a line, while that of 2 lines is a point.

The above predicates can be efficiently supported by using quadtree as the underlying spatial data structure. A spatial join degenerates to a spatial selection if all but one of the spatial arguments are fixed. Also, spatial operators, selections, and joins become simple spatial predicates which are either true or false if all the arguments are bound.

In addition to these new built-in predicates, $\mathcal{LDL}$++ also allows the definition of new geometric datatypes and predicates in C++. This gives users the ability to develop specialized geometric algorithms that operate on the spatial data structures efficiently.

4 Example Application

In this section, we show a few example queries in a cyclone tracking application. We have 2 tables in our database,

```
{cyclone( Id:inetger, X:float, Y:float ) }
```

which is updated as cyclones appear, move, and dissipate, and

```
{cyclone_track( Id:integer, Track:[(time,float,float)] )}
```

which stores cyclone tracks as lists of 3-dimensional points in space and time.

Query 1: Identify the location of formation of cyclones.

```
creation_location(ID, X, Y) <-
        ep(+cyclone(U)), ID = U.id, X = U.x, Y = U.y.
```

Query 2: Identify all pairs of cyclones that come within 50 miles of LA within a month of each other.

```
cyclone_pair(ID1, ID2) <-
        ep( [ -+cyclone(X), distance(X.x, X.y, 'LA', D1),
              -+cyclone(Y), Y.id ~= X.id,
              distance(Y.x, Y.y, 'LA', D2),
              D1 <= 50, D2 <= 50,
              Y.evtime - X.evtime <= one_month
            ]
        ), ID1 = X.id, ID2 = Y.id.
```

Query 3: Find cyclone tracks which starts within a certain time interval and region (Vindow), and which last longer than a certain range and reach minimum pressure below a given millibar value.

```
find_track( Id, Track, Intrv, Range, Window, SLP ) <-
        cyclone_track( Id, Track ),
        startDuring( Track, Intrv ),
        startInWindow( Track, Window ),
        rangeGT( Track, Range ),
        minPressureLT( Track, SLP ).

startDuring( [(Time,_,_)|_], Intrv ) <-
        contain( Intrv, Time ).

startInWindow( [(_,Lat,Lon)|_], Window ) <-
        contain( Window, pt(Lat,Lon) ).
```

```
rangeGT( Track, Expected_range ) <-
        range( Track, Actual_range ),
        Actual_range >= Expected_range.

minPressureLT( Track, SLP ) <-
        minPressure( Track, Min_SLP ),
        Min_SLP < SLP.
```

where range, and minPressure are aggregate predicates in $\mathcal{LDL}$++.

5 Architecture

Our objective is to build a prototype system testing various techniques and demonstrating the feasibility and benefits of semantic processing of atmospheric and earth sciences data. This processing combines searches on alphanumeric data (such as annotations and system generated keywords describing images), summary data (such as 3-D contours and iconic abstracts) and pixel-based images. The most effective query-execution strategies in such a multimedia database intermingles searches in the different media domains to exploit their interrelationships and reduce the search space [2]. Therefore we propose the architecture as shown in Figure 3.

The advantages of this architecture include practicality and flexibility, obtained by combining the strengths of various database systems and prototypes available at UCLA. The Postgres extended relational database system[16] is used to store, search and manage alphanumeric information. At the core of the system there is an integration layer which coordinates the underlying systems, and supports high-level functions such as spatio-temporal reasoning on multimedia domains. This integration layer currently uses the $\mathcal{LDL}$++ system for spatial temporal queries. In addition, $\mathcal{LDL}$++ has an open architecture towards other DBMSs. These features allow efficient and flexible access to multimedia data stored in underlying databases, as needed to realize this architecture. Of particular importance, e.g., is the ability of taking full advantage of externally defined ADTs such as those provided in Postgres. Image and spatial data, managed by existing GISs, will also be connected and used in a similar fashion.

Another important function of such integration is the extraction of index

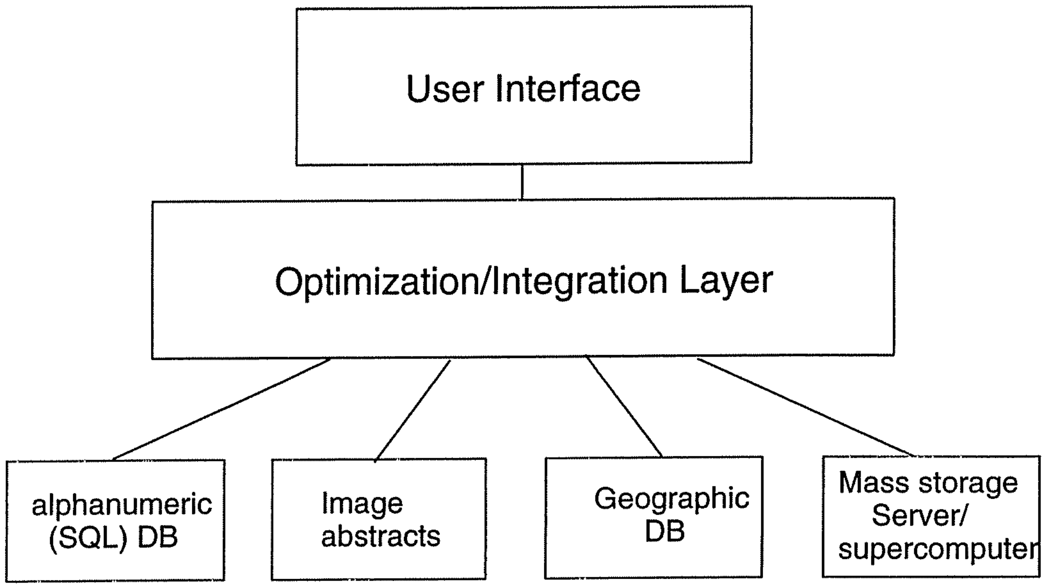

User Interface
Optimization/Integration Layer
alphanumeric (SQL) DB
Image abstracts
Geographic DB
Mass storage Server/ supercomputer

Figure 3: Architecture for Scientific DB Testbed

terms, characteristic shapes and similar summary data to be used for object indexing. This *data ingestion* process is basically a bottom-up operation that feeds summary data to the upper layer and to the Alpha-numeric DB and Shape DB. Due to the variety of applications and data types that we would want to support it is clear that such a system must be extensible and allow users to provide the "transducers" that extract the indexing features for an object. The extraction of the index entries can be computation intensive and will be executed on a supercomputer platform.

6 Conclusion

In this paper, we have described the uses of a logic based data language ($\mathcal{LDL}$++) in a prototypical atmospheric science application. Spatio-temporal extensions to $\mathcal{LDL}$++ allow scientists to easily query and reason about spatio-temporal information. In addition, query processing is enhanced by the use of active content-based indexes that are supported by $\mathcal{LDL}$++ in a straightforward manner.

There are still many issues that need to be addressed by the database and GIS communities to make temporal GIS commonly available. In particular, query processing and optimization[3, 5], support of multiple level storage [15, 17], and parallelism[6] are a few areas that we are currently pursuing.

References

[1] A. Arakawa, and V. R. Lamb, "Computational Design of the Basic Dynamic Processes of the UCLA General Circulation Model." Methods in Computational Physics, 17, Academic Press, 1977.

[2] W. G. Aref, and H. Samet, "Extending a DBMS with Spatial Operations." In *2nd Symposium on Large Spatial Databases*, 1991.

[3] W. G. Aref, and H. Samet, "Optimization Strategies for Spatial Query Processing." In *Proceedings of the 17th VLDB Conference*, Barcelona, Spain, 1991.

[4] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, and C. Zaniolo, "The $\mathcal{LDL}$System Prototype." In *IEEE Transaction on Knowledge and Data Engineering*, vol. 2, no. 1, 1990.

[5] T. Y. Leung, "Query Processing and Optimization in Temporal Database Systems." Ph.D. thesis, Department of Computer Science, University of California, Los Angeles, 1992.

[6] T. Y. Leung, and R. Muntz, "Temporal Query Processing and Optimization in Multiprocessor Database Machines." In *Proceedings of the 18th VLDB Conference*, Vancouver, British Columbia. Canada, 1992.

[7] C. R. Mechoso, C. C. Ma, J. D. Farrara, and J. A. Spahr, "Simulations of Interannual Variability with a Coupled Atmosphere-Ocean General Circulation Model." In *the fifth Conference on Climate Variations*, American Meteorology Society, Boston, MA, 1991.

[8] S. Naqvi, and S. Tsur, "A Logical Language for Data and Knowledge Bases." Computer Science Press, New York, NY, 1989.

[9] G. Phippis, M. A. Derr, and K. A. Ross, "Glue-Nail: A Deductive Database System." In *Proceedings of 1991 ACM SIGMOD International Conference on Management of Data*, 1991.

[10] R. Ramakrishan, D. Srivastava, and S. Sudarshan, "*CORAL*: A Deductive Database Programming Language," In *Proceedings of the 18th VLDB Conference*, Vancouver, British Columbia. Canada, 1992.

[11] H. Samet, "The Design and Analysis of Spatial Data Structures." Addison-Wesley, Reading, MA, 1990.

[12] T. R. Smith, "Towards a Logic-based Language For Modeling and Database Support in Spatio-Temporal Domains." In *Proceedings of the 5th International Conference of Spatial Data Handling*, 1992.

[13] R. Snodgrass, and I. Ahn, "A Taxonomy of Time in Database." In *Proceedings of ACM SIGMOD Conference on Management of Data*, Austin, TX, 1985.

[14] S. M. Sripada, "A Logical Framework for Temporal Deductive Databases." In *Proceedings of the 14th VLDB Conference*, Los Angeles, CA, 1988.

[15] M. Stonebraker, "Managing Persistent Objects in a Multi-level Store." In *Proceedings of 1990 ACM SIGMOD International Conference on Management of Data*, Denver, CO, 1990.

[16] M. Stonebraker and G. Kemnitz, "The POSTGRES Next Generation Database Management System." In *Communication of ACM*, vol. 31, no. 10, 1991.

[17] M. Stonebraker, "An Overview of the Sequoia 2000 Project." In *Proceedings of the 1992 COMPCON Conference*, San Francisco, CA, 1992

[18] C. Zaniolo, "Intelligent Databases: Old Challenges and New Opportunities." In *Journal of Intelligent Information Systems*, 1, Kluwer Academic, 1992.

Developing Applications with CORAL*

Amitabh Saran, Keith Park, Yongmao Chen, Ana Paula de Aguiar
Terence R. Smith, and Jianwen Su

Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
{saran, bahk, yongmao, anap, smithtr, su}@cs.ucsb.edu

1 Introduction

We are involved in a cooperative study with a group of EOS [1] investigators at the University of Washington relating to the development and implementation of spatially-distributed models of water, sediment and solute transport in the Amazon river basin. A detailed examination of the requirements and computational activities of these investigators revealed that adequate computational support for the formulation and iterative construction/testing of their models was severely lacking [3]. The overall goal of our project is to design and develop a system providing computational support that would permit these and other investigators to achieve their scientific goal more efficiently [2, 3]. We believe that the efficiency of such individuals is measured greatly in terms of how quickly they can find appropriate solutions to problems in their domain of application. Our research involves understanding the nature of earth science investigations and subsequently the design and development of a Modeling and Database System (MDBS) [4] to provide explicit, high-level support for model development and database construction, maintenance and access. The focus of the current paper is on the study of logic based languages appropriate for implementing applications that involve spatial data and scientific computations. One such application is the problem of water routing in the Amazon basin which is discussed in detail.

In the modeling of scientific phenomena, an activity of fundamental importance, is the organization of knowledge into *conceptual domains (C-domains)*, which consist of collections of entities and transformations across entities. An important goal of scientific activity is the discovery of appropriate *representational domains (R-domains)* for these C-domains. In particular, we may view scientists as employing lattices of

*Supported in part by NSF grant IRI-9117094.

domains, in which relatively *high-level* C- and R- domains are defined inductively in terms of *low-level* domains; e.g., low-level domains like *polygons* and *line segments* may be used to define high level domains like *Digital Elevation Models (DEM)* and *Drainage Basins.* Each R-domain is implemented by one *abstract R-domain* and a set of isomorphic *concrete R-domains.* Abstract R-domains provide specifications of the elements and transformations, while concrete R-domains define the structures of the elements and realize the transformations. Languages used by scientists should allow them to construct and apply an extensible and potentially very large collection of R-domains and transformations, thus generating a "complete" set of concepts that they need for modeling some set of phenomena. Such languages should be largely declarative in nature. Coupled with the knowledge that '*algorithm = logic + control*' we were prompted to use logic programming languages in our implementation. The use of logic programming languages allows the possibility of not only describing, but also of coding and optimizing, the implementations of a system in a uniform framework. We chose the deductive database language CORAL [5] as our basis.

In this paper, we briefly report our experiences of using CORAL in our scientific applications. A large set of *R-domains* was built in CORAL. Higher level domains were constructed from primitive domains, thus creating a domain hierarchy. In general, R-domains form a lattice in which inheritance of both structure and transformations occurs naturally. Also, a simplified version of an application of major importance to Earth scientists was coded in CORAL. It involved the construction of a drainage network from a DEM, and the application of a spatially-distributed, time-sliced hydrological model in order to compute the discharge of runoff from the network. We found that logic programming based languages are good for applications which are not computationally intensive. Mixing some control primitives (e.g., pipelining, or other annotations in CORAL) into pure logic or relational programming languages improves the performance to a significant degree. Finally, coupled with imperative languages such as C++, logic programming languages appear capable of providing a reasonable basis for developing applications which can even be computationally intensive. Along with the MDBS project, we have also designed a model for managing the metadata of R-domains and transformations, which we represent in the same database of CORAL relations. This allows queries on the metadata to be expressed in CORAL.

This paper is organized as follows. The development of the complex spatial abstract and concrete R-domains and their associated transformations, together with the Amazon watershed example is the focus of Section 2. Section 3 describes the experiences of coding in CORAL. Conclusions are presented in Section 4.

2 Application Development in CORAL

R-domains have been used to formulate the data model for MDBS, where a C-domain is represented by (a) a single abstract R-domain that corresponds to the C-domain and its associated transformations and (b) a set of isomorphic concrete R-domains, each of which implements the abstract R-domain and its transformations in terms of a specific, but distinct, representation. Polygons, for example, can have several concrete representations such as sequences of points or sequences of line segments.

2.1 Spatial Domains

A library of spatial domains with their associated transformations is created in CORAL. The characteristics used to define a concrete R-domain include

1. the domain name,
2. the structure or representation of the elements in the domain, e.g., set, tuple etc.
3. constraints on the values of domain elements, e.g., constraints that differentiate a convex polygon from the general class of polygons etc.,
4. a set of transformations for elements of the domain.

The library includes primitive domains such as *point* with X- and Y- coordinates, *point_pairs* and further complex domains such as *polygons*, *rectangles* and *rasters.* Transformations on these created domains also form part of the library. For example, the following code represents the detection of colinear points when the domains *point* and *line* have been defined.

```
pt_colinear(Point1, Point2, Point3) :-
        Line = line(Point1, Point2),
        pt_is_on_line(Point3, Line).

pt_is_on_line(Point, Line) :-
        Point = point(X,Y),
        Line = line(point(X1,Y1),point(X2,Y2)),
        Y = ((Y2-Y1)/(X2-X1))*(X-X1) + Y1.
```

An important mechanism for the inductive construction of new R-domains involves the application of constructors (e.g., line in the above program, or set, tuple, etc.) to elements from previously defined domains, which may also have constraints defined on their values. It is worthwhile to note that such constraints induce the inheritance structure on the R-domains, e.g., defining the domain *convex polygon* from the domain *polygon* by placing constraints on the structure of a polygon.

2.2 Water Routing Computation

We used CORAL to code an application problem which we are investigating with a group of EOS investigators. One of the basic problems which these earth scientists are facing is to construct computational models which represent the routing of water, sediment and solutes down the whole of the Amazon River watershed. This

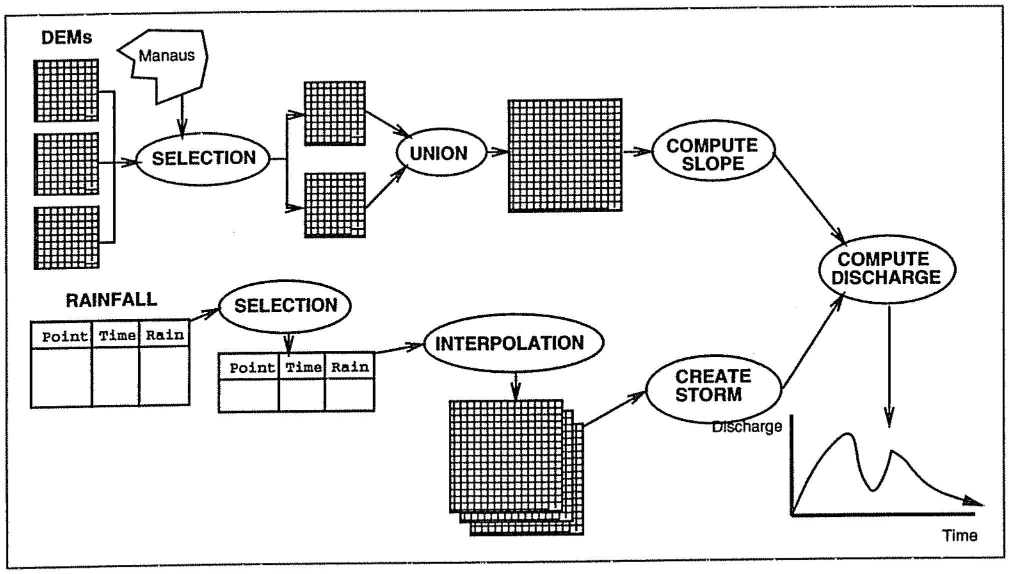


Figure 1: Simplified View of the Discharge Computation

problem is relatively complicated in terms of the structure of the various domains and transformations and the size of the datasets that are required for an adequate analysis.

In Figure 1, we show the datasets and operations that provide a simplified view of the problem. Each dataset corresponds to an element in some concrete R-domain and the operations correspond to transformations on domain elements. The sequence of computations involves first choosing DEMs of interest by intersecting them with an appropriate area (region around Manaus) and then combining them into a single DEM. Rainfall data from points inside this area and for specific time periods are retrieved and interpolated over the DEM. The DEM is used to generate an "isochrone map" that models the flow of water from each point in the DEM to the mouth of the river. *Hydrographs* is a domain of representations of the discharge of water at the mouth as a function of time.

DEMs are represented as a grid structure with tuples comprising pairs of point coordinates and their corresponding elevations. These tuples are stored as ground facts in CORAL. Each point on the grid has eight points adjacent to it, the *neighbors*. For example, the code for the *neighbor* relation would be:

```
neighbor(point(I,J), point(M,N)) :-
        dem(point(I,J), _), dem(point(M,N), _),
        I-1 <= M, M <= I+1,
        J-1 <= N, N <= J+1.
```

Water flows from a point to its neighbor having the least elevation. The mouth of the DEM/watershed is the point from which there is no further drainage. The isochrone map or the "equal travel distance map" which consists of lines connecting points located equally reachable from the mouth of the basin, is thus generated. For generating the flow path from each point in the DEM to the basin mouth, we have:

```
final_flow(point(From_X1, From_Y1), point(To_X1, To_Y1)) :-
        one_flow(point(From_X1, From_Y1), point(Next_X1, Next_Y1)),
        final_flow(point(Next_X1, Next_Y1), point(To_X1, To_Y1)).
```

where `one_flow` generates facts corresponding to the flow between adjacent nodes in the DEM.

Rainfall data at observation points are recorded at fixed intervals of time. In other words, we have layers of data, each corresponding to a time instant, containing rainfall measurements for all the observation points. These layers are termed *Rain Rate Field Layers (RRFL)*. For computational purposes, the data are organized in the form of relations with individual tuples consisting of the time of recording, coordinate of the observation point and the rainfall measurement at the particular instant of time. This data, from observation points located within the DEM area (Manaus), are to be interpolated over the DEM, thus generating the *Grid Rain Rate Field Layers (GRRFL)*. The process of interpolation may be denoted as :

$$I_{GRRFL}(t,p) = \frac{\sum_q (I'_{RRFL}(t,q)/dist(p,q))}{\sum_q (1/dist(p,q))}$$

where

$I_{GRRFL}(t,p)$	: interpolated intensity at point p for time slice t,
$I'_{RRFL}(t,q)$	: recorded intensity at observation point q for time slice t,
$dist(p,q)$	: Euclidean distance between points p and q, and
$\sum_q$	: represents the sum over all observation points q in the *rain rate layer.*

Finally, the discharge at the mouth of the watershed is computed by taking the isochrone map and the interpolated rainfall layers generated by the previous modules. The corresponding CORAL code is:

```
discharge(Del_dist, Time_n, sum(<Rate>)) :-
                        Dist = Del_dist * Time_n,
                        iso_points(Pt_d, Pnts),
                        Pt_lyr = Dist - Pt_d + 1,
                        grrfl(Dist, Pt_lyr, Pnts, Rate),
                        Pt_d <= Dist.
```

where

`Del_dist`	: Measure of computation step, and
`Time_n`	: Time instant at which discharge is to be computed.

This is the runoff at the mouth of the basin as a function of time.

The whole exercise is important in modeling the quantity of water flow at various points in the Amazon watershed as a result of rainfall in the region. Storms can be localized and the impact of each of them on the flow can be estimated. It can be used to predict future events based on results of collected data.

2.3 Metadata Management and User Interface

We also use CORAL to manage the metadata effectively within the database and to implement, in part, a simple user level interface. We briefly explain the use of CORAL in these two aspects.

Metadata management is necessary in scientific applications. Users often need to search for appropriate datasets/algorithms from databases to design their computation models. In our database, metadata provides the description of the abstract and concrete R-domains, their structures and transformations. We store the metadata in four kinds of relations in CORAL. They are:

1. the *abstract* relation for storing characteristics of the abstract R-domains: the names, sub/super-domains relationships, and the corresponding transformations;
2. the *concrete* relation which describes all the concrete representations for each abstract domain, the structural representations, the constraints on the elements in the domain and the transformation implementations;
3. an *element* relation for each abstract domain describing the elements; and
4. the *isomorphism* relation containing transformations between different concrete domains of the same abstract domain.

For example, an abstract R-domain for DEM would be described as a tuple:

```
abstract
(AbsDomain, ConDomains, Transformations, SupDomains, SubDomains).
abstract(dem, {dem_PointElevPair, dem_Binary},
              {[elev_at, [dem,point], [elevation]],
               [eight_neighbor, [dem,point], [pointSet]]},
              {raster},
              -).
```

Each *element* relation can store information relating to the nature of the dataset defining the element. This includes basic *header* information which is used commonly (as part of the Spatial Data Transfer Standards) for describing the contents of binary files containing *Landsat Images* or DEMs. The information includes the resolution of data, the coordinates defining the boundaries and the name of the place etc. Another important field, *Lineage*, can also be incorporated to reflect the error propagation at various stages of computation and element creation from existing datasets.

Metadata can be queried regarding the existence of elements, domains, transformations and the interrelations across domains. One important feature for using CORAL for defining the schema is its ease of use compared to other database systems. Tables can be defined declaratively and data, elements, tuples/facts can be added incrementally without concerns for restrictions on buffer size. Populating tables is as simple as editing a text file. All column boundaries and type checks are taken care of by CORAL at the time of query evaluation.

We have also experimented with CORAL in implementing a small set of user level commands that allow easy manipulation of databases. We use predicate names in CORAL to represent functions that are performed by a code segment. Such "keywords" can be incrementally added to represent compositions across functions by using existing predicates. Thus, while the command set is rigid and terse it is easily extendable. An example is the command apply, which takes as input a function (transformation) and applies it to the elements of a concrete R-domain. Suppose we wish to compute the perimeter of a polygon. It is the sum of all the line segments forming the polygon. The code segment pgon_perimeter is shown in the following.

```
pgon_perimeter(Polygon, Perimeter) :-
        Polygon = polygon(PointSeq),
        ptSeq_to_lsegSeq(PointSeq, LineSegSeq),
        apply_to_all(lseg_length, LineSegSeq, LengthSeq),
        list_sum(LengthSeq, Perimeter).

apply_to_all(_, [], []).
apply_to_all(Function, [Input|IList], [Output|OList]) :-
        apply(Function, Input, Output),
        apply_to_all(Function, IList, OList).
```

Here the function apply_to_all applies the given function to the input list. Similarly, for computing the area of a polygon, we use function composition by employing the signed sum of the area of individual trapezoids constituting the polygon:

```
pgon_area(Polygon, Area) :-
        apply_sequence([pgon_signed_area, num_abs], Polygon, Area).

pgon_signed_area(Polygon, SignedArea) :-
        pgon_to_trapezoidSeq(Polygon, TrapezoidSeq),
        apply_to_all(trapezoid_signed_area, TrapezoidSeq, AreaSeq),
        list_sum(AreaSeq, SignedArea).
```

We feel that this feature i.e., creation of new functions using low-level primitive ones, is very crucial in developing large applications.

3 Observations

CORAL has provided our first exposure to the use of a logic programming language in developing a large scale application. The facts that CORAL was a new language and that the declarative style of programming is different from the imperative, was initially a source of problems. At the time of our application development, CORAL was still in a premature stage of development (version 0 and the first cut of version 1). All features had not been implemented. Presumably CORAL has evolved into a much more stable system since then. Our main effort is to share the experiences of using a logic programming language for implementing scientific applications involving spatial databases. We summarize the following four aspects: declarativeness, performance, improvements, and useful (expected) features.

3.1 Declarativeness

Its declarative nature makes CORAL fairly easy to work with. The online help is useful for the beginners. The constructs are similar to those of an imperative language (e.g., assignment, expression evaluation).

It opens a new domain of working for scientists. Being largely declarative in nature, logical phenomena can be modeled with relative ease. It is our estimate that many scientist teams focus up to 50% of their time on computational issues that are irrelevant to their scientific research [4]. We believe that a large part of this is consumed in management of files, input/output, memory, data structures and the like, many of which can be implicitly taken care of when we use CORAL. Thus, scientists can focus their attention on *what* phenomena need to be modeled rather than *how* it should be done.

3.2 Performance

Comparing its performance to imperative languages would not be doing justice to CORAL. The difference becomes apparent when dealing with CPU-intensive computations, e.g., while generating the isochrone map (isolines), working with DEMs of relatively small sizes ($100 \times 100 = 10,000$ point grid). But then, computing the transitive closure over a large number of points would be equally intensive for all logic languages.

One important feature of CORAL is its integration with C++, thus supporting both the declarative and imperative programming styles. CORAL commands can be embedded in C++ to make an "extended" C++. This proves extremely useful for CPU-intensive work, e.g., for isolines module, the drainage network was ascertained in an imperative fashion using C++ and subsequent rainfall computations were computed in a declarative style without breaking the relation abstraction. We found that having the support of an imperative language is not only of value for purposes of efficiency, but is actually *necessary* for coding computational tasks. In our application, estimating the isochrone map was the bottleneck, taking almost 30 minutes (in the run time of about 31 minutes for the total application) for a small $50 \times 50 = 2,500$ point DEM grid. The module was about half of the application in terms of the size of code, and extremely CPU-intensive. However, when we coded the same computation in C++, with embedded CORAL, the same task just took 11 seconds (without any optimization). We could have optimized on the CORAL program, but still any figure in minutes for a DEM as small as 2,500 points, is not comparable to imperative languages.

3.3 Improvements

We achieved improvement without using C++ by generating ground facts from declarative modules on a one-time basis, e.g., the neighboring points and the detection of flow from a point to an adjacent point, can be precomputed and remains unchanged for a particular DEM. This greatly reduced the execution time since most of the computation of transitive closure was eliminated.

Annotations provided the tools to make the programs run efficiently. Since they can be used to provide directives on a per-module basis, different annotations can be used at places according to the requirements. We were able to increase efficiency to a considerable extent using annotations like *pipelining*, permitted speedups of almost 10-14% for some modules. Some annotations that could have helped us, had not been implemented in the initial versions of CORAL.

3.4 Other Useful Expected Features

One feature that the initial versions of CORAL lacked was a stable database support for persistent relations. While operating on large DEMs, domains and all associated transformations with only in-memory support, we faced problems regarding lack of free space.

The usefulness of CORAL in our application would have increased if it were possible to have the CORAL engine running as a backend query processor to which commands could be piped or sent using streams. This could lead to the integration of CORAL with other software packages if required.

Finally, debugging features in CORAL posed a problem while the application was being developed. The initial versions were not equipped with an interactive debugger, so error detection during execution was not simple.

4 Conclusion

Using a logic-based, declarative language leads to a culture shock for programmers experienced in using imperative languages. The concepts of variable definitions and iterations is replaced by predicates, facts and recursion. For beginners, not worrying about the underlying memory management and access methods, is in itself, a big difference. As far as our project was concerned, CORAL provided the necessary infrastructure for development, but what it lacked was support for persistent relations. The initial versions were not equipped with a very stable link with the EXODUS [6] data management system, so loading spatial domains and transformations at each stage became a bottleneck. For the project, the ideal use of CORAL came as part of metadata management. It was very well designed, easy to use and largely extensible.

Acknowledgements

We would like to acknowledge the help of Raghu Ramakrishnan in using CORAL that lead to more efficient programs. We are also thankful to Praveen Seshadri for his contribution in system related problems.

References

[1] EOS: A Mission to Planet Earth, NASA, Washington, D.C., 1990

[2] T. R. Smith, J. Su, D. Agrawal, and A. El Abbadi. Database and Modeling Systems for the earth sciences. In IEEE Bulletin of the Technical Committee on *Data Engineering*, 16(1), March 1993. (Special Issue on Scientific Databases).

[3] T. R. Smith, J. Su, T. Dunne, A. El Abbadi, and D. Agrawal. Requirements for modeling and database systems supporting investigations of large scale flows in the amazon watershed, 1993

[4] T. R. Smith, J. Su, D. Agrawal, and A. El Abbadi. MDBS: A modeling and database system to support research in Earth sciences. *Proc. of the NSF Scientific Database Projects*, AAAS Workshop on Advances in Data Management for the Scientist and Engineer, Boston, MA., February, 1993.

[5] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudershan. CORAL - Control, Relation and Logic. *Proceedings of the 18th VLDB Conference*, Vancouver, Canada, 1992.

[6] Michael Carey, David DeWitt, Joel Richardson, and Eugene Shekita. Object and file management in the EXODUS extensible database system. *Proceedings of the International Conference on Very Large Databases*, August 1986

A Declarative Language Environment for Knowledge-Workers

Oris Friesen

Bull Information Systems
P.O. Box 8000, MS: H32
Phoenix, AZ 85066-8000
friesen@system-m.az05.bull.com

Abstract

There have been numerous attempts to identify applications that benefit from and that exploit the unique features of declarative languages, in general, and deductive logic, in particular. This paper outlines a scenario in the data mining, or knowledge discovery, application domain that has been defined by potential users in the "real world." The scenario is illustrated using a pseudo-language that has been developed in an advanced development project at Bull Information Systems. The scenario described here would seem to be applicable to a large number of problems in many different application areas.

1. Introduction

For some time researchers have been casting about for appropriate application areas for the deployment of declarative and deductive logic database systems. Most of these attempts to identify candidate application domains have been conducted by the researchers and developers of the systems [Tsur, Kris]. If deductive logic database systems are to become popular in the marketplace, it is mandatory that "real world" prospective users of those systems (e.g., end-users, application developers, knowledge workers) become involved in identifying their advantages for given application domains. This paper is an example of such an endeavor, and as such, it is not a research paper.

This paper describes an actual situation in which the declarative components of a Deductive and Object-Oriented Database (DOOD) system are applied to solve information requirements that exist today. It is important to note that these requirements have been identified by the prospective users, not by the system developers. These requirements cannot be satisfied by current database management systems (DBMSs) in the marketplace.

The application domain consists of a part of the public school system in the United States. The DOOD system described here is the product of an Advanced Development Project at Bull Information Systems and is not available as a commercial product. This system will be referred to in this paper as NEXT-GENERATION.

This paper represents the point of view of a knowledge worker: the user of the DOOD system. Its purpose is to demonstrate the real-world applicability of DOOD systems in general, and of declarative logic in particular. Section 2 discusses the background and sets the context for focusing on the public school application domain. Section 3 discusses the nature of data mining. Section 4 introduces the knowledge-worker environment provided by NEXT-GENERATION in which the public school officials can operate. Section 5 develops a scenario that utilizes a declarative logic language to perform data-mining activities. Finally, Section 6 discusses the advantages provided by this approach and why this approach is superior to using features provided by existing systems, such as SQL-based DBMSs.

2. Background

As public school districts in the United States move toward site-based management and increased local control, a number of new issues have been identified. Many of these issues are motivated by the need of school officials to access and evaluate various kinds of socioeconomic data in the school neighborhood. One of the most serious problems from a database perspective is how to integrate a collection of databases and apply them in ways that are dramatically different from their intended use. For example, school officials often must resort to importing data by hand into spreadsheets and performing rather crude and simplistic analyses based on elementary binary relationships among data elements. Not only do the various installed data models not support the new needs, the current state of the technology does not support their principal tasks: data mining and knowledge discovery.

The situation is ripe for demonstrating support for data mining kinds of activities. Additionally, it is paramount in such an environment to integrate (on the desktop) various commodity tools, such as spreadsheets, Computer Aided Design (CAD) tools and Geographical Information Systems (GIS).

This paper discusses a specific program that has been established to provide assistance to the local school districts in one part of the United States. To protect the privacy of this organization, it will be referred to in this paper as the Inter-School Program (ISP). The data available comes from many government sources at the federal, state, and local levels. The agencies involved in providing data span almost all socioeconomic services and support provided to the community. The information includes census data (i.e., demographics), criminal data, educational performance data, and social services and assistance data (e.g, welfare). The objective of the program is to make this data available to school administrators (such as school principals and superintendents) to help them formulate policy and make decisions based on a comprehensive and collective view of each individual's data.

The school administrators that have provided input, have evidence that among the myriad volumes of data is knowledge upon which they can base decisions. They acknowledge that they generally start out having no specific queries in mind, but after browsing through the data, queries tend to suggest themselves. They have already successfully demonstrated this to a limited degree, using spreadsheets and hard-copy paper reports. But they are clearly in need of more sophisticated and refined sets of supporting software tools.

The problem definition is quite open-ended due to the administrators' limited understanding of what lies within the data. It appears that their understanding of what they can extract from the data is based on nothing more than conjecture and speculation. Yet, they clearly have the domain knowledge to cooperatively guide the search. This is a classical case of data mining (or knowledge discovery) and is where NEXT-GENERATION and deductive logic programming can play a role.

3. Data Mining

What is data mining? It is a term used to suggest the discovery of knowledge not implicitly stored in the database being queried. It is usually used in conjunction with data that is not collected for any particular purpose or at least not for the purpose the questioner has in mind. The data volumes are usually very large, and the sources are frequently quite diverse and often of an historical nature (e.g., census data, complete project data compiled from numerous subcontractors, DNA sequence data, etc.).

The process of data mining usually begins with a poorly formed notion of the task to be performed, which can be called an hypothesis [Tsur]. This hypothesis is refined through iterative steps during which human judgment is employed by analyzing the results and determining whether or not further refinement is necessary. If no more refinement is found to be necessary, then in the judgment of the human user, the task has been completed. The result of the task is that the hypothesis has been confirmed, disproven or remains unproven.

The problem until now has been how to process and analyze the data. Users have been forced to use different and sometimes incompatible tools (such as spreadsheets, DBMSs, etc.) or to develop complex and resource-expensive specialized applications. With the emergence of declarative logic-based systems, these users have an opportunity to analyze their data in a much more consistent and powerful fashion.

4. A DOOD Environment for Knowledge-Workers

The deductive technology of NEXT-GENERATION derives from the EKS System prototype [Baye], but it incorporates object-oriented features both at the architectural and language levels. In addition to a data model with objects and values resembling that used in the O2 system, it supports a data manipulation language with a Datalog-based declarative component (to write deduction rules and integrity contraints) and a more classical imperative component (to write methods and functions). It is built on a storage manager having many features in common with object-managers, rather than with more traditional relational data stores.

Without going into a detailed description of NEXT-GENERATION (because of proprietary considerations), some of its basic features are described here. The knowledge-worker who uses NEXT-GENERATION has some acquaintance with deductive logic. The core of a NEXT-GENERATION schema consists of types and predicates. A type is a named set of values. Types are made up of some basic internally-defined data types (such as integer, boolean, character, etc.) plus user-defined types, which are sometimes referred to as abstract data types. User-defined types can be created with the help of NEXT-GENERATION-provided type constructors, which allow one to create complex collections and structures, such as tuples, sets, lists, bags.

Predicates define relationship patterns between NEXT-GENERATION values. The pieces of information stored in a NEXT-GENERATION database are the facts built up as instances of the predicates. A predicate closely resembles a relation or a class.

A basis predicate is characterized by having a self attribute, of type ref (ref is a

NEXT-GENERATION-specific data type), such that the value of this self attribute uniquely identifies the corresponding (explicit) fact throughout the lifetime of the database (i.e., it plays the role of an object identifier). Basis predicates are updatable. (A fact for a basis predicate is also referred to as an object and is the means whereby object-orientation can be supported by NEXT-GENERATION.) Facts for a basis predicate can be explicitly stored as part of the content of a database, or they can be derived by the rules that define this basis predicate. Below is an example of a basis predicate:

```
City   (ref                Self,
        string[20]         Name,
        string[30]         Country,
        integer4           Population)
```

A virtual predicate does not have a self attribute. Thus facts built up as instances of a virtual predicate do not have a fact identifier. Facts corresponding to a virtual predicate can only be derived (i.e., they cannot be explicitly stored). Virtual predicates are not updatable. They can be thought of as being similar to views in a relational system. Below is an example of a virtual predicate:

```
City   (string[20]         Name,
        string[30]         Country,
        integer4           Population)
```

The NEXT-GENERATION knowledge-worker operates in a client/server environment. The NEXT-GENERATION tools (modeling tools, query tools, etc.) are available from the desktop personal computer, and the data and schema definitions reside on a server system that can maintain and manage multiple databases.

5. A Scenario

Below is a typical scenario that might be followed by a knowledge-worker in a public school environment who has access to the ISP data and to the NEXT-GENERATION system. The first-person narrative style is used below to better simulate a real-world situation.

I am the school principal of a high school in some large city in the United States. Lately I have become concerned over the possible increase in drug use within my school district. My fears have been motivated by rumors and unconfirmed reports from students and teachers. If these reports are true, I would like to confirm them and identify the responsible parties. I would especially like to know if there is a core of students who might be soliciting or recruiting other students into this pattern of behavior and in what part of the district they reside.

So, the challenge is to uncover some pattern that will allow me to make an informed assessment. First, I need to understand what data I have available for examination. Then, I need to formulate some meaningful hypotheses and test them against the available data.

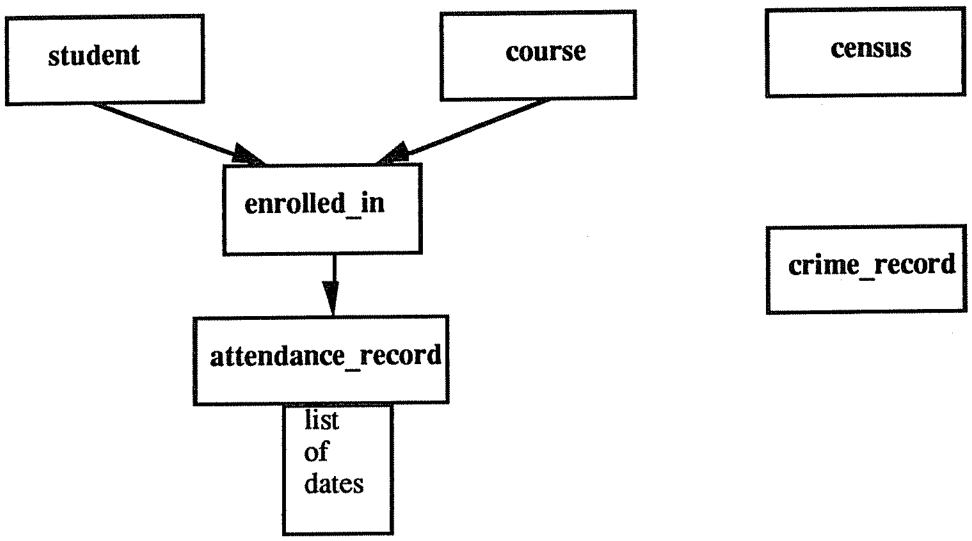
The data comes from three separate sources provided through the ISP. Each source deals with a different data group. One group consists of criminal justice data dealing with crimes in the urban area in which my school is located. Another group contains public schools data for the school district that includes my school. It has data such as student grades, age, addresses and attendance history. The last data group consists of census track data.

The criminal justice data consists of the following sort of information: subject's name,booking data including date and location and the category of the cause for booking (e.g., drug-related, vehicular offense).

The public schools data is the most extensive, and it can be categorized into three logical groupings: student data, course data and attendance data.

The census track data contains household information gathered as part of the most recent census for the area in which my school is located.

The following picture contains an approximation of the schema that is managed by NEXT-GENERATION. The data common to a course in which a student is enrolled (such as "grade received") has been factored out and placed in the enrolled_in predicate.



The data below contains an approximation of the schema content that will actually be displayed by NEXT-GENERATION.

student:	school_id, student_id, name tuple (last, first, middle), sex, address tuple (street number, street name, apartment number, city, state, zipcode), phone_no, birth_date tuple (year, month, day), student_home_room, student_grade_level, student_special_education_needs, guardian tuple (last, first, middle).
course:	course_id, course_name, section_name, section_period_number, room_number, instructor_name tuple (last, first, middle), year_and_semester.
enrolled_in:	ref enrollee, ref course, grade_received.
attendance_record:	ref enrolled_in, list (tuple date tuple(year, month, day), period, semester, instruction_day_number, present_flag, excused_flag, ill_flag, tardy_flag, explanation text).
crime_record:	booking_id_number, subject_name tuple (last, first, middle), booking_date tuple (year, month, day), booking_loc, statute_violation_category.
census_data:	household_address tuple (street number, street name, apartment number, city, state, zipcode), school_district.

The following data contains a sample of the NEXT-GENERATION pseudo-syntax for some of the persistent predicate schema definitions.

```
        declare persistent type year_month_dayType tuple {
                year            integer2,
                month           integer1,
                day             integer1
        };

        declare persistent type full_nameType tuple {
                last_name               varstring[32],
                first_name              varstring[32],
                middle_name             varstring[32]
        };

        declare persistent type addressType tuple {
         house_number           varstring[10],
         street_name            varstring[32],
         apartment_number       varstring[6],
         city                   varstring[32],
         state                  string[2],
         zip_code               string[10],
        };
```

The student predicate contains information about each student

```
declare persistent basis predicate student (
        school_id                               string[5],
        student_id                              string[8],
        name                                    full_nameType,
        sex                                     string[1],
        address                                 addressType,
        phone_no                                string[8],
        birth_date                              year_month_dayType,
        student_home_room                       string[3],
        student_grade_level                     string[2],
        student_special_education_needs         string[2],
        guardian                                full_nameType
);
```

Now I want to formulate a working hypothesis and test it against the data. I suspect that students may be "cutting" classes to sell drugs to one another or to share drugs with each other. I would like to find out if that is true. If it is true, then I would like to find out which students are responsible for this activity. So, as an overall hypothesis, I am going to assume that there is some relationship between unexcused absences from classes and drug use. I also assume that there is some data stored in the databases that would allow me to validate this hypothesis.

I proceed as follows:
1. First, I formulate an hypothesis.
2. Then, I translate that hypothesis into a query.
3. Then, I execute the query and observe the results.
4. If the results do not verify or invalidate the hypothesis, I will reformulate or refine the query and go to step #3.

Since we are going to deal with drug use, we need to have some way of defining who qualifies as a drug user. Therefore, my first hypothesis will need to deal with how to define an actual user of drugs.

So, I hypothesize that a drug user is someone who has a criminal record and has been arrested for a drug-related crime.

When I translate this into a NEXT-GENERATION rule, I get the following syntactical definition (for the sake of simplicity let us say that the category for drug-related offenses is encoded as the numeric constant "750"):

```
declare persistent derived predicate drug_user (name full_nameType) {
        drug_user (name N)
        <- crime_record (subject_name N, statute_violation_category 750)
};
```

I execute the above statement and examine the results.

On examining the results, I notice that I have retrieved a very large number of people in the crime_record database (i.e., all of those who have committed drug-related offenses). Moreover, I realize that I do not know which ones of these are students. So, I need to reformulate my hypothesis.

I subsequently hypothesize that a drug user (for my purposes) is a student who has a criminal record and has been arrested for a drug-related crime.

Translating this into an NEXT-GENERATION rule, I get the following definition in NEXT-GENERATION syntax:

```
declare persistent derived predicate drug_user (name full_nameType) {
        drug_user (name N)
                <-      student (name N)
                        and crime_record (subject_name N,
                                statute_violation_category 750))
};
```

I execute the above statement and examine the results. This time I am satisfied with the results, so I have a working definition for "drug user."

Now, I need to form an hypothesis that will help me to define a potential "recruiter" for drug usage and/or purchase. I now know (from the last query executed) that a small number of students have actually been arrested for drug use. It is reasonable to suspect that other students have been influenced by these users. If so, then these other students have probably been "hanging around," or associating, with the drug users. Before I can confirm this suspicion, we would like to identify those students who have had some "contact" with the users identified above.

The question becomes, how to define that "contact." Perhaps (I hypothesize) such "contact" can be defined by a relationship that focuses on course periods that are missed by some set of students. I decide to pursue this line of reasoning.

The next question becomes one of deciding how to use the attendance data that is available to me. I know that I have an attendance record for each course in which every student has been enrolled for every day of instruction.

I first need to identify those students (and the associated date) who have missed at least one course period with an unexcused absence. The pseudo-syntax for such a derived fact is:

```
declare persistent derived predicate gone_one_period (
        name                    full_nameType,
        absent_date             year_month_dayType,
        absent_period           integer2
)
{gone_one_period (name N, absent_date D, absent_period P) distinct
        <-      student (self S, name N)
                and enrolled_in (self E, enrollee S)
                and attendance_record (enrolled_in E,
                                present_flag[X] FALSE,
                                excused_flag[X] FALSE,
                                date[X] D,
                                period[X] P
)
};
```

After executing the above statement and examining the results, I have the name, date and period number for all unexcused absences. It occurs to me that here may be a hidden pattern here that is not explicitly stored in the databases. That is, I would like to know if the students who have been “cutting” classes have had contact, either direct or indirect, with those students we have identified as drug users above.

By direct contact, I mean those who have personally associated with the identified drug users. By indirect contact, I mean students who have personally associated with students who have personally associated with students who have personally associated with students . . . who have personally associated with the identified drug users. (This is a typical example of a recursive query.) I will call this set “candidate_at_risk_students.”

Next, I need to define what I mean by a “candidate_at_risk_student.” In other words, what constitutes “being in contact with” or “associating with?”

I will define this as follows: Those students who have been absent without an excuse on the same day and for the same course period as a drug user will be defined as an “at-risk student.” Moreover, those students who have been absent without an excuse on the same day and for the same course period as another “candidate_at_risk_student” will themselves be classified as “candidate_at_risk_students.”

First, we want to define a rule that we can use to identify companions. We will define companions as those pairs of students who have missed one course period together.

```
declare persistent derived predicate companion (
        name                    full_nameType,
        date_together           year_month_dayType,
        period_together         integer2,
        associated_with         full_nameType
)
{companion (name C, date_together D, period_together P, associated_with A)
        <-      gone_one_period (name C, absent_date D, absent_period P)
                and gone_one_period (name A, absent_date D, absent_period P)
                and not equal (A, C)
};
```

Now, retrieve all students (call them “candidate_at_risk_students”) who have missed 1 course period either with a drug_user or with another candidate_at_risk_student. We need to be careful here, because we want to retrieve these associations in a temporal manner. That is, if name1 misses a class with name2 on date1, then we are interested in who else missed a class with name2 after date1.

```
declare persistent derived predicate candidate_at_risk_student (
        name                    full_nameType,
        date_together           year_month_dayType,
        period_together         integer2,
        associated_with         full_nameType
)
{candidate_at_risk_student (name C, date_together D, period_together P,
                                associated_with A)
        <-      companion (name C, date_together D, period_together P,
                                associated_with A)
                and drug_user (name A);
candidate_at_risk_student (name C, date_together D, period_together P,
                                associated_with A)
        <-      candidate_at_risk_student (name C1, date_together D1,
                                period_together P,  associated_with A)
                and companion (name C, date_together D2, period_together P,
                                associated_with C1)
                and greater_than (D2, D1)
 };
```

I know that some students attend external schools, such as "magnet" schools and others are transferred to different schools for various reasons. I am curious about how many of the candidate_at_risk_students live within the school boundaries of the school which they attend. In order to perform this query I need to access the census track data.

```
declare persistent derived predicate in_district (
        name                    full_nameType,
        home_address            addressType,
        school_id               string[5],
        household_address       addressType
)
{in_district (name I, home_address A, school_id S)
        <-      candidate_at_risk_student (name I)
                and student (student_id I, school_id S, address A)
                and census_data (household_address A, school_district S)
};
```

It suddenly strikes me that a far more interesting query would be to find out how many candidate_at_risk_students attend schools in districts other than the one to which their household belongs. So I define an "outsider" predicate using the concept of negation simply by adding the keyword "not" to the above rule and changing the predicate and rule names from "in_district" to "outsider.".

```
declare persistent derived predicate outsider (
        name                    full_nameType,
        home_address            addressType,
        school_id               string[5],
        household_address       addressType
)
{outsider (name I, home_address A, school_id S)
        <-      candidate_at_risk_student (name I)
                and student (student_id I, school_id S, address A)
                and not census_data (household_address A, school_district S)
};
```

This demonstrates how easily one can form the negation of a given query within the NEXT-GENERATION syntax. I now have a list of candidate_at_risk_students who are going to schools outside of their home districts. This may give me some insight into how external influences are affecting student behavior.

6. Summary and Discussion

The above scenario helps to illustrate that deductive logic and declarative programming certainly have a role to play in the world outside of academia. The kind of data mining activity discussed above is generally applicable to many domains beyond that of the public school system. It is especially useful in areas where the data comes from numerous sources and is arranged in a flat, or relational, format, because the relationships that exist among such data elements are generally hidden and are ripe for discovery by a DBMS that can exploit the semantic power of logic.

The scenario also illustrates the usefulness of features such as recursion and negation. In fact, negation, as used in the last stage of the above scenario, is a very natural notion to end-users and knowledge-workers. It is normal to pose a query and then, when the answer is not revealing (or perhaps is too voluminous), to pose the negation of the same query.

It is well known that recursion is not supported by existing SQL-based systems, so a good deal of the above scenario would have required some host language procedural code if attempted in SQL. Although SQL does support a "not in" construct, the correct use of negation in SQL is far from intuitive. To perform the negation scenario outlined above would require some rather complex SQL syntax.

There are many other logic database concepts that will prove to be useful to today's users in the commercial marketplace. This paper has tried to illustrate a few of them, but much more remains to be done. What is most important is that, as we try to identify areas of applicability for deductive logic, it is imperative to involve the potential users in the "real-world," such as the ISP users who have provided the input to the scenario described above.

Acknowledgements

I would like to thank Laurent Vieille for his assistance and many constructive comments in preparing and reviewing this paper.

References

[Baye] Bayer, P., A. Lefebvre and L. Vieille, "Architecture and Design of the EKS Deductive Database System," submitted for publication.

[Kris] Krishnamurthy, R. and T. Imielinski, "Research Directions in Knowledge Discovery," *SIGMOD Record*, Vol. 20, No. 3, September 1991, pp. 76-78.

[Tsur] Tsur, S., "Data Dredging," *Data Engineering*, Vol. 13, No. 4, December 1990, pp. 58-63.

Programming the PTQ Grammar in XSB

David S. Warren[*]

Department of Computer Science

SUNY at Stony Brook

Stony Brook, NY 11794-4400

October 5, 1993

Abstract

The XSB language is an implementation of the Prolog programming language that supports tabling. As such it is an implementation of SLG-def resolution, a subsystem of SLG resolution [CW93]. It can also be seen as an integration of OLDT evaluation into Prolog. The resulting language is significantly more declarative than Prolog, allowing the programmer to concentrate on the specification of the problem and devote less time to avoiding infinite loops and resolving other procedural issues as required in Prolog.

In 1973 Richard Montague published an influential paper on formal semantics for natural language, entitled *The Proper Treatment of Quantification in Ordinary English*, or *PTQ* for short [Mon74]. In that paper he gave a formal grammar for a fragment of English, a formal logical language IL, and a translation from English sentence derivations to IL, thereby providing a logical semantics for English sentences in his fragment. In [War79], I described an implementation of the *PTQ* framework, including a parser, translator and logic simplifier. That implementation was done in the LISP programming language.

In this paper I describe a re-implementation of the *PTQ* system in XSB. The original implementation of *PTQ* in LISP took me several years; the re-implementation in XSB took several days.

1 Introduction

Prolog has long been recognized as a powerful language for natural language processing. Indeed, the needs of natural language processing constituted a major part of Alain Colmerauer's original motivation for the design and implementation of Prolog [CKPR73]. However, when actually using Prolog for processing natural language grammars, several limitations become apparent. One claim often made by Prolog proponents is that by programming in Prolog, one gets the parser "for free". This has some truth if one can stay completely within the Definite Clause Grammar (DCG) formalism and use the recursive descent parsing one obtains from Prolog. (The DCG formalism is provided through a simple preprocessor in most Prolog systems and supports a very simple and

[*]Supported in part by the National Science Foundation under Grants No. CCR-9102159 and USE-9150417.

elegant way of writing annotated grammars.) However, for larger projects, one rather quickly finds that recursive descent parsing is seriously deficient, with its limitations requiring avoidance of left-recursive grammars and careful left-factoring to avoid exponential recomputation. So for large projects one is forced to write one's own parser, introducing considerable computational overhead, and thereby losing much of the power and original attractiveness of Prolog. We note, however, that unification *is* retained, and that can also be a very powerful and helpful tool.

The XSB system is an implementation of Prolog that supports tabling. It is an implementation of a subsystem of SLG resolution [CW93] and a variant of OLDT resolution [TS86]. Because it uses tabling, it can correctly process left-recursive grammars. Indeed it will correctly and finitely handle all context-free grammars. As Prolog's evaluation strategy results in a recursive-descent parsing algorithm when applied to DCG's, so XSB's tabling strategy results in a variant of Earley's parsing algorithm [Ear70]. So with XSB, one can use the DCG formalism directly and get an Earley parser "for free". This makes XSB very attractive for implementing many grammars.

In [Mon74] Richard Montague presented a grammatical system that showed how one could provide a reasonable logical semantics for an interesting subset of English. He was interested in problems of pronouns and their antecedents, and wanted to provide a complete account of their semantics. He was also interested in certain philosophical problems present in most logic-based treatments of natural language semantics. For example, most logic-based systems, given a sentence such as "John seeks a unicorn", would generate its meaning to be something like "There is an X such that unicorn(X) and seeks(John,X)." However, this seems hardly an adequate representation of its meaning, since the English sentence can be true and yet there be no unicorns, whereas the logical statement, when true, implies the existence of a unicorn. The system that Montague gave in *PTQ* showed how such problemmatical sentences could be handled in a formal logic without encountering such nonintuitive results.

The *PTQ* system includes a grammatical component that provides an inductive definition of a set of English sentences. Montague did not give this definition in any particular grammar formalism, but just as an inductive definition of a set of strings. To provide a language for specifying the meanings of these sentences, Montague gave the syntax and semantics of a formal logical language, which he called *Intensional Logic* (IL). IL is a complex modal variant of type theory. To provide meanings for the English sentences, he mapped them to formulas in IL. This was done by a recursive definition over the definition of English sentences: for every word of a basic category in the syntax, he provided an IL formula; and for every syntactic rule that combined English phrases to form larger phrases, he provided a semantic rule that combined the corresponding IL subformulas to form an IL formula for the larger phrase.

Several of the syntactic rules in *PTQ* are naturally left-recursive. For example, rule S10 combines an IV phrase with a adverb to create another IV phrase, as in combining the IV phrase, "walk in the park," with the adverb, "slowly," to obtain the IV phrase, "walk in the park slowly." This is most naturally modeled with the left-recursive context-free rule: $IV \longrightarrow IV\ IAV$. Also, rules of conjunction are left recursive, as are the most natural representations of the rules of quantification. Of course, it is always possible to transform a context-free grammar with left recursion into another grammar that recognizes the same language but does not contain left recursion. So one possibility would be to transform Montague's rules to eliminate their left recursion. However, since the semantic rules correspond one-for-one with the syntactic rules, it is highly desirable to use the syntactic rules as they are. Any modification to the syntactic rules would require a corresponding change to the semantic rules. One could argue that such a change would really modifying Montague's system, and therefore such an implementation would be implementing some other system, *not PTQ*.

For these reasons we must process the left recursive grammar as it is. These considerations led me in [War79] to propose and implement in LISP a tabling interpreter for Augmented Transition Network grammars, a formalism popular at the time for representing complex grammars. I implemented the *PTQ* syntax and semantics using that system. Later, after coming to learn of Prolog, it was immediately clear to me that the tabling would apply to Prolog, which eventually led to the XSB system. Pereira and Warren in [PW80] argue cogently why Prolog is a better formalism than ATN's for representing complex grammars. Now with XSB, a Prolog system that can handle the left-recursion of *PTQ*, it seemed appropriate to reconsider the implementation of *PTQ* using DCG's. This paper describes the results of this endeavor.

The paper is structured as follows: Section 2 gives a brief introduction to the syntax of *PTQ* and describes the grammar. Section 3 briefly describes the logic IL, its representation and algorithms for simplification, and gives the rules of translation from *PTQ*. Section 4 gives examples and timings for processing certain sentences under both an OLDT meta-interpreter and the XSB system. Section 5 concludes.

2 PTQ Syntax

In *PTQ* Montague gave a rather complex inductive definition of sets of phrases of various syntactic categories. Rather than repeating that definition here, the reader interested in the details is referred to [Mon74]. Here I will simply give the resulting grammar, glossing over the difficulties enountered in reformulating these rules into a form for which a parser exists. For details of this reformulation the reader is referred to [FW78, War79].

Here we give an intuitive description of how PTQ treats pronouns and antecedents by describing an analogy between English sentences and programs written in a statically scoped programming language, such as Pascal. So we are here using Pascal as an analogy to English (as Montague perceived it.) Consider an analogy between program variables in Pascal and noun phrases in English. In Pascal, each program variable must be explicitly declared. That declaration determines a scope, in which all occurrences of the declared identifier refer semantically to that same program variable. In English, as Montague saw it, the analogue of a program variable is a term (or noun phrase). But rather than using the same identifier each time the semantic object is referenced, the full term is used only the first time; subsequently a pronoun is used. For example, in Pascal, one might say "X := Y+1; Y := X;", and in English one might say, "John loves Mary, and she loves him." In the Pascal fragment, the two occurrences of the X refer to the same program variable; in the English fragment, the words "John" and "him" refer to the same semantic object. So the problem in parsing English, under PTQ, is to find the proper coreferents. Also, of course, terms are not declared in English, as program variables must be in Pascal, so the parser must also find the *scope* of the term. In PTQ, the scope can be a sentence, an intransitive verb phrase, or a common noun phrase, and the parser must find all the possibilities.

It is clear that such a grammar for English would be highly ambiguous. Montague intended his grammar to produce all logically possible structures, given a sentence. It was viewed as the job of some other (unspecified) component to determine which of several (or many) parses is most likely intended in any given context.

The parser maintains a symbol-table-like structure, called the *store*. The store, which is associated with a nonterminal, contains a list of the terms in the portion of the string spanned. It also contains a representation for each pronoun. Then at points at which there may be a "declaration", the store is scanned for terms and pronouns that might be coreferent. For those that are found, a substitution rule (Montague's counterpart to a declaration rule) is applied and the participating term and pronouns are deleted from the store.

The DCG rules are given in Figures 1 and 2. Each nonterminal has at least two attributes: the first in which the parse tree is returned, and the last which contains the *store*.

The DCG rules are labeled by the names Montague gave to the clauses in his inductive definition of the syntax. There were rules S1-S17. (S17 dealt with past and future tenses of sentences, and we have not included it here.) The typical structure of a rule first has calls to parse the constituents, then an append to combine the stores returned for the constituents, and finally a call to st/3 which simply constructs the parse tree for the current nonterminal using the parse trees of the constituents. The nonterminal symbols beginning with "b" (as in biv/2 or bte/2) represent "basic" categories, and interface to the lexicon (not shown here.) They always recognize words.

Consider the sentence, "John loves a woman and she loves him." One of the two parses for this sentence is shown in Figure 3. Parsing "John" uses the third clause for te/4. It calls tel/4 to get a term, in this case the basic term (bte) "John", and then puts it in the store and returns "trace(_)" as the parse. Similarly "loves Mary" is parsed using the S5 rule for iv/3. These constituents are put together using the S5 rule for s/2. "she loves him" is parsed similarly, but using the second clause for te/4 to recognize the pronouns and put them in the store. The second s/2 rule, implementing Montague's S11, conjoins these two phrases and concatenates their stores. Then rule S14 implemented by the last clause for s/2, applies twice, once to remove "John" and "him" from the store, and once to remove "a woman" and "her". At this point, the variables representing the semantic objects are unified, indicating for example that the subject of the first subsentence, trace(_), and the object of the second subsentence, pro(_), are coreferent, with both variables becoming A. (The predicate get_term_pros/3 finds and removes a term and some following pronouns, identifying their variables.) These substitution rules construct the final parse. The fact that these pairs may be removed in either order is what causes there to be two parses for this sentence.

There are two features of this grammar worth noting here. First, it is simple. The rules of the grammar (after a little study) are really a very straightforward translation of Montague's inductive rules. Second, this grammar would *not* execute under Prolog. There are several instances of left-recursive rules: the conjunction rules for sentences, terms, and intransitive verb phrases; the rule adding an adverb to an intransitive verb phrase; the rule for adding a relative clause to a common noun phrase; and, perhaps most importantly, the subtitution rules S14–S16. Reformulating this into a grammar without left recursion would certainly complicate the grammar immensely.

```
% St is the Store: a list of records term(Te,Var,Gen), pro(Var,Gen)

s(T,St) -->      % S4
  te(Te,sub,_G,St1), iv(Iv,s,St2),
    {append(St1,St2,St), st(T,s4,(Te,Iv))}.
s(T,St) -->      % S11
  s(S1,St1), word(and), s(S2,St2),
    {append(St1,St2,St), st(T,s11a,(S1,S2))}.
s(T,St) -->      % S11
  s(S1,St1), word(or), s(S2,St2),
    {append(St1,St2,St), st(T,s11b,(S1,S2))}.
s(T,St) -->      % S14
  s(T1,St1),
    {get_term_pros(term(Te,X,_Gen),St1,St), not_occurs_in(X,(Te,St)),
     st(T,s14,(X,Te,T1))}.

iv(Iv,T,[]) --> biv(Iv,T).  % S1
iv(Iv,T,St) -->    % S5
  btv(Tv,T), te(Te,obj,_G,St),
    {st(Iv,s5,(Tv,Te))}.
iv(Iv,T,St) -->    % S12
  iv(Iv1,T,St1), word(and), iv(Iv2,T,St2),
    {append(St1,St2,St), st(Iv,s12a,(Iv1,Iv2))}.
iv(Iv,T,St) -->    % S12
  iv(Iv1,T,St1), word(or), iv(Iv2,T,St2),
    {append(St1,St2,St), st(Iv,s12b,(Iv1,Iv2))}.
iv(Iv,T,St) -->    % S10
  iv(Iv1,T,St1), iav(Adv,St2),
    {append(St1,St2,St), st(Iv,s10,(Iv1,Adv))}.
iv(Iv,Tn,St) -->    % S7
  bivt(Vb,Tn), s(T,St),
    {st(Iv,s7,(Vb,T))}.
iv(Iv,T,St) -->    % S8
  biviv(Vb,T), iv(Iv1,i,St),
    {st(Iv,s8,(Vb,Iv1))}.
iv(Iv,T,St) -->    % S16
  iv(Iv1,T,St1),
    {get_term_pros(term(Te,X,_Gen),St1,St), not_occurs_in(X,(Te,St)),
     st(Iv,s16,(X,Te,Iv1))}.
```

Figure 1: DCG for *PTQ* (Sentences and Verb Phrases)

```
iav(Adv,[]) --> biav(Adv).  % S1
iav(Adv,St) -->    % S6
  pr(Pp), te(Te,obj,_G,St),
    {st(Adv,s6,(Pp,Te))}.

te(Te,C,G,St) --> te1(Te,C,G,St).
te(Te,C,G,[pro(X,G)]) -->   % for S3, S14-S16
  pro(_Pro,G,C),
    {st(Te,pro,X)}.
te(Te,C,G,St) -->           % for S14-S16
  te1(M,C,G,St1),
    {st(Te,trace,X), append(St1,[term(M,X,G)],St)}.

te1(Te,_C,G,[]) --> bte(Te,G).
te1(Te,_C,G,St) -->    % S2
  det(Det), cn(Cn,G,St),
    {st(Te,s2,(Det,Cn))}.
te1(Te,C,G,St) -->    % S13
  te(Te1,C,G,St1), word(or), te(Te2,C,_,St2),
    {append(St1,St2,St), st(Te,s13,(Te1,Te2))}.


cn(Cn,G,[]) --> bcn(Cn,G).
cn(Cn,G,St) -->    % S3
  cn(Cn1,G,St1), word(such), word(that), s(T,St2),
    {delete_some(pro(X,G),St2,St3), append(St1,St3,St),
     st(Cn,s3,(X,Cn1,T))}.
cn(Cn,G,St) -->    % S15
  cn(Cn1,G,St1),
    {get_term_pros(term(Te,X,G),St1,St), not_occurs_in(X,(Te,St)),
    st(Cn,s15,(X,Te,Cn1))}.
```

Figure 2: DCG for *PTQ* (Adverbs, Terms, and Nouns)

```
s14
  A
  john
  s14
    B
    s2
      a
      woman
    s11a
      s4
        trace(A)
        s5
          love
          trace(B)
      s4
        pro(B)
        s5
          love
          pro(A)
```

Figure 3: Parse for "John loves a woman and she loves him"

3 PTQ Semantics

For Montague, the syntax of PTQ was the least interesting part. It existed only to support the semantics, which we describe here. Again we will not go into great detail (I spent several years trying to understand the implications of these definitions), but refer the interested reader to the original source.

The logical language Montague used as the target for his translation of English he called IL. We will give the syntax of IL using the symbols we use in our ASCII representations in our implementation. The logic is typed, so first is the definition of the types:

1. e and t are types, the types of individuals and the type of truth values, respectively.
2. Whenever A and B are types, then $< A, B >$ is a type, the type of functions from objects of type A to objects of type B.
3. Whenever A is a type, then $< s, A >$ is a type, the type of functions from possible worlds to objects of type A.

The definition of the meaningful expressions of type a of IL (ME_a) is as follows:

1. Variables are identifiers with capital initial letters; constants are identifiers with lowercase initial letters. Each constant and variable has a type (which we leave implicit in our representation).

```
'walk' ==> X\ (walk @ (*X))
'John' ==> P\ ((*P) @ (^john))
'love' ==> P\ X\ (*P @ (^(Y\ (love @ (*Y)) @ (*X))))
'a' ==> P\ Q\ exists(X,(*P @ X /\ *Q @ X))
```

Figure 4: Some Translations of English Words to *IL*

2. If $\alpha \in ME_a$ and U is a variable of type b, then $U\backslash\ \alpha \in ME_{<b,a>}$, (the λ-abstraction rule where \ is the λ operator and written in infix notation.)
3. If $\alpha \in ME_{<a,b>}$ and $\beta \in ME_a$, then $\alpha @ \beta \in ME_b$, (the rule of function application.)
4. If $\alpha, \beta \in ME_a$, then $\alpha = \beta \in ME_t$.
5. If $\phi, \psi \in ME_t$, and U a variable, then $not(\phi)$, $(\phi\ /\backslash\ \psi)$, $(\phi\ \backslash/\ \psi)$, $(\phi -> \psi)$, $exists(U, \phi)$, $all(U, \phi) \in ME_t$, (the sentential connectives and quantifiers.)
6. If $\alpha \in ME_a$ then $(\hat{}\alpha) \in ME_{<s,a>}$, (the intension of α.)
7. If $\alpha \in ME_{<s,a>}$ then $(*\alpha) \in ME_a$, (the extension of α.)

The semantics of these meaningful expressions is given through a model theory, which we will not describe here. In this model theory certain operations on meaning expressions preserve their meanings. In particular, the λ-calculus operations of α-reduction and (a minor variant of) β-reduction preserve meanings.

The translation of English sentences into meaningful expressions in IL begins with the translation of English words. In *PTQ* many of the English words translate directly to constants in IL. These constants tend to have a rather complex type, but because of the so-called "Meaning Postulates", many of these constants are logically equivalent to meaningful expressions of a much simpler type. So we translate the words to these simpler-typed expressions. Examples of the translations of words are given in Figure 4.

For example, the English word 'walk' is translated into a function that takes an intensional object and sees whether the logical constant *walk* is true of it. The English word 'John' translates to a set of propositions true of the intension of the individual *john*. The English article 'a' translates to, essentially, an existential quantifier.

For each syntactic rule, there is a semantic rule that creates meaningful expressions for compound phrases by combining the meaningful expressions of its constituents. Figure 5 gives these semantic rules as represented in our Prolog program.

These rules just construct more complex expressions from simpler ones. For example, the rule for *s4* creates the meaning of a sentence from the meanings of a subject and a verb phrase, by applying the meaning of the subject to the intension of the meaning of the verb phrase. For example the meaning of "John walks" is obtained from this rule by combining the meanings of "John" and "walks", resulting in: `(P\ ((*P) @ (^john))) @ (^(X\ (walk @ (*X))))`. By λ-reducing this, we can obtain: `walk @ (*john)`. (We also applied the identity $(*(\hat{}X)) \equiv X$.)

```
st(Te,s2,(Det,Cn)) :- Te <== Det@ ^Cn.
st(Cn,s3,(X,Cn1,T)) :-  Cn <== X\ (Cn1@X /\ T).
st(T,s4,(Te,Iv)) :- T <== Te@ ^Iv.
st(Iv,s5,(Tv,Te)) :-  Iv <== Tv@ ^Te.
st(Adv,s6,(Pp,Te)) :- Adv <== Pp@ ^Te.
st(Iv,s7,(Vb,T)) :- Iv <== Vb@ ^T.
st(Iv,s8,(Vb,Iv1)) :- Iv <==  Vb@ ^Iv1.
st(Iv,s10,(Adv,Iv1)) :- Iv <== Adv@ ^Iv1.
st(T,s11a,(S1,S2)) :- T <== S1 /\ S2.
st(T,s11b,(S1,S2)) :- T <==  S1 \/  S2.
st(Iv,s12a,(Iv1,Iv2)) :-  Iv <==  X\ (Iv1@X /\ Iv2@X).
st(Iv,s12b,(Iv1,Iv2)) :- Iv <== X\ (Iv1@X  \/ Iv2@X).
st(Te,s13,(Te1,Te2)) :- Te  <==  P\ (Te1@P  \/ Te2@P).
st(T,s14,(X,Te,T1))  :- T  <== Te@ ^(X\T1).
st(Cn,s15,(X,Te,Cn1)) :- Cn <== Y\Te@ ^(X\ (Cn1@Y)).
st(Iv,s16,(X,Te,Iv1)) :- Iv <== Y\Te@ ^(X\ (Iv1@Y)).
st(Te,trace,X) :- Te <== P\ *P@X.
st(Te,pro,X) :- Te <== P\ *P@X.

X <== X.       % all translations
```

Figure 5: Semantic Rules for *PTQ*

These Prolog rules execute to construct the "direct translation" of any English sentence (or phrase) in the grammar. All we needed to do was change the definition of the predicate *st/3* to make it construct semantic representations (i.e., expressions in *IL*) rather than syntactic ones (i.e., parse trees.)

These rules when executed generate a direct translation for every parse of a sentence. For example, for our sentence "John loves a woman and she loves him", the program generates two direct translations:

```
(A\* A@^ j)@^ (B\(C\D\exists(E,* C@E/\* D@E))@
  ^ woman@^ (F\((G\* G@B)@^ ((H\I\* H@
  ^ (J\love@* J@* I))@^ (K\* K@F))/\(L\* L@F)@
  ^ ((M\N\* M@^ (O\love@* O@* N))@^ (P\* P@B)))))

(A\B\exists(C,* A@C/\* B@C))@^ woman@^ (D\(E\* E@^ j)@
  ^ (F\((G\* G@F)@^ ((H\I\* H@^ (J\love@* J@* I))@
  ^ (K\* K@D))/\(L\* L@D)@^ ((M\N\* M@^ (O\love@* O@* N))@^ (P\* P@F)))))
```

Needless to say, these are not particularly enlightening. It also happens to be the case that they are logically equivalent and can be shown to be so simply by λ-reducing and replacing an expression of the form *(^(M) with the logically equivalent M. We have defined a predicate in XSB, *lred/2*, to do this. Applying *lred* to these gives us just one answer:

```
exists(A,woman@A/\(love@* A@j/\love@j@* A))
```

which is much more readable, saying much more perspicuously that there is an X such that X is a woman and John loves X and X loves John.

It turns out that this property of there being several direct translations that all λ-reduce to the same meaningful expression is very common. That is, the grammar is very highly ambiguous syntactically, but not so ambiguous semantically. One strategy is to generate direct translations and then λ-reduce them all at the end to find the unique meanings; another is to λ-reduce each meaningful expression as it is constructed, and return only reduced expressions. This has the possibility of greatly reducing the parsing ambiguity by reducing the combinatorial explosion of multiple parses. For example, by reducing two parses to one in each of two parallel constructions, we reduce the number of parses that need to be constructed on the higher level from four to one.

In order to modify the program to do the reductions on each internally constructed expression, we need only change the definition of `<==/2`. We change it to:

```
X <== Y :- lred(Y,X).
```

With this definition, reduction is done at each step as meanings are constructed. Note that the elimination of the duplicates generated by common reductions is done automatically by the underlying tabling mechanism of the XSB system.

In the next section we look at how this reduction in number of parses impacts performance.

4 Evaluation and Statistics

We have not done a comprehensive study of the performance of the PTQ grammar executing under the XSB system. We have run a few examples in several ways to get a general idea of the costs and tradeoffs of processing PTQ sentences under XSB. There are two kinds of comparisons that we can do. Firstly, we can compare XSB with an alternative implementation of the same grammar. We no longer have the Lisp program of [War79] to compare with. Also, we have been unable to get this program to compile and run under the CORAL system [Ram90], but this may be due to the early version we tried. We can, however, run the Prolog program developed here on another evaluator. We had previously written a metainterpreter in Quintus Prolog that does tabling, a system we called the XOLDT system. With only a few additional declarations, the XOLDT metainterpreter will execute the PTQ grammar. Thus we can compare the times for processing various sentences under XSB evaluation of the program and under XOLDT metainterpretation of the program.

Secondly we can compare the times to execute the grammar to create all the direct translations for a sentence with times to generate all distinct meanings of the sentence. As described above, by λ-reducing the meaningful expressions representing the meanings of each subcomponent as it is constructed, we end up generating only distinct meanings. We have run these two different ways of processing meanings under the XSB system.

We have used three sentences as our initial examples. The sentences are:

1. John believes that Mary wishes to walk in a park.
2. Mary believes that John finds a unicorn and he eats it.
3. John tries to find a unicorn and wishes to eat it.

(Sentences from Montague's fragment do tend to be rather strange.)

Sentence	No. of Dir. Trans.	No. of. Red. Trans.	Time for OLDT dir	Time for XSB dir	Time for XSB red
1.	167	9	26.8	7.5	2.5
2.	42	3	82.0	14.2	3.0
3.	5	1	1.45	0.65	0.60

The examples are run on a Sparc2 and the times given are in seconds. Several comments are worth making. The huge number (167) of direct translations for the first sentence arises from the fact that there are three noun phrases that can have many scopes and orders. The other two have fewer direct translations since the pronouns constrain the possible scopes of the noun phrases.

In comparing the times taken to get the direct translations using the XSB engine with those using the XOLDT metainterpreter, we should note that only 5 predicates are tabled. Much of the computation is being done directly in Prolog, and the XOLDT metainterpreter calls compiled Prolog for Prolog code. Since Quintus Prolog is several times faster than XSB Prolog, the fact that XSB is this much faster than the metainterpreter on the PTQ grammar is impressive. Also, note that indeed doing the λ-reduction on each subsentential expression does drastically reduce the number of answers generated and decreases the execution time.

5 Conclusion

The algorithm for processing PTQ described in this paper is not exactly the one of [War79]. In this implementation, the store and its associated information is passed up the parse tree only. This can be seen in the grammar rules: the *appends* all follow the grammar symbols and build the tables to return to the caller. In [War79] and in the first XSB program for *PTQ*, some information was passed downwards. Information concerning potential antecedents was passed down the tree so that when a pronoun was encountered, the system could determine its antecedents. Passing this information down resulted in more table entries and therefore less sharing. So to see if passing information up was better, I recoded the algorithm. With the XSB system, I was able to change the algorithm in about a day. Actually, it was the unification of Prolog that made this so easy. In LISP, I wanted to make the decision of an antecedent at the point of seeing the pronoun, so I could co-index the pronoun and the antecedent and put that index in the parse tree at that point. In Prolog, with unification, I did not need to have the index at the time I put it into the tree; I could simply put a variable into the parse tree and later at a higher point in the parse, when I determine the antecedent, I just unify the index and the variable in the tree. Of course, this could have been programmed in LISP, but it would have required significantly more programming effort.

The question remains as to how this relates to Deductive Databases. Note that this program requires some of the more advanced features of deductive DBs, such as handling of variables and structures. But these have been discussed in the DDB community, e.g. in CORAL. (We did try to see if this grammar could run under CORAL, but were unsuccessful. More effort and knowledge of CORAL than we had available at the time would be necessary.) Also, the more traditional features of databases (and hopefully of deductive databases), such as an ability to store and maintain large volumes of persistent data, would come in handy for large natural language systems. For example, realistic lexicons for natural language can be very large and must be indexed, and must be updatable.

A final note is that the tension between top-down and bottom-up processing that is currently being worked out in the database community also existed in the area of grammars and language theory. And the solutions seem to be similar.

References

[CKPR73] Alain Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. Un système de communication homme-machine en Francais. Technical report, Groupe Intelligence Artificielle, Université Aix-Marseille II, 1973.

[CW93] Weidong Chen and David S. Warren. Query evaluation under the well-founded semantics. In *Proceedings of the Twelfth Symposium on Principles of Database Systems.* ACM, 1993.

[Ear70] J. Earley. An efficient context-free parsing algorithm. *CACM*, 13:94–102, 1970.

[FW78] Joyce Friedman and David S. Warren. A parsing method for Montague grammars. *Linguistics and Philosophy*, 2:347–372, 1978.

[Mon74] Richard Montague. The proper treatment of quantification in ordinary English. In Richmond H. Thomason, editor, *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, 1974.

[PW80] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278, May 1980.

[Ram90] Raghu Ramakrishnan. The CORAL deductive database system. In Jan Chomicki, editor, *Proceedings of the NACLP Workshop on Deductive Databases*, Aug 1990. Austin, TX.

[TS86] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings of the Third International Conference on Logic Programming*, pages 84–98, 1986.

[War79] David S. Warren. Syntax and semantics in parsing: An application to Montague grammar. Technical report, Department of Computer and Communication Sciences, The University of Michigan, Ann Arbor, Michigan, 1979.

AMOS: A Natural Language Parser written in *LOLA*

Günther Specht Burkhard Freitag Heribert Schütz

Institut für Informatik, Technische Universität München
Orleansstr. 34, D-81667 München, Germany

email: {`specht,freitag,schuetz}@informatik.tu-muenchen.de`

In this paper we present the set-oriented bottom-up parsing system AMOS which is a major application of the deductive database system *LOLA*. AMOS supports the morpho-syntactical analysis of old Hebrew and has now been operationally used by linguists for a couple of years. The system allows the declarative specification of Definite Clause Grammar rules. Due to the set-oriented bottom-up evaluation strategy of *LOLA* it is particularly well suited to the analysis of language ambiguities.

1 Introduction

In this paper the set-oriented bottom-up natural language parsing system AMOS, a major application of the deductive database system *LOLA* [3, 2], is presented. The AMOS system serves for the morpho-syntactical analysis of old hebrew text and is intensively used by linguists. A grammar for old Hebrew [12] has been formalized as a Definite Clause Grammar (DCG) and represented as a *LOLA* program. The Definite Clause Grammar formalism and the evaluation of the corresponding logic programs by a PROLOG interpreter are wellknown [11, 10]. Definite Clause Grammars have been intended to provide a means for the *declarative* specification of grammar rules which can directly be interpreted as a logic program. PROLOG-based DCG parsers, though, suffer from a number of drawbacks

- Left-recursive rules can not be interpreted directly.
- Backtracking involves the duplicate construction of syntactic structures.
- In case of ambiguity of the grammar rules only one parse tree at a time is constructed thus introducing another source of backtracking.
- At least if information beyond the word position within a text is required the dictionary has to be stored as a collection of unit clauses. The connection of a PROLOG based system with its tuple-at-a-time strategy to a set-oriented relational database needs a buffering interface. This problem is known as the impedance mismatch problem.

Solutions have been proposed for each of the problems listed above. In [10] bottom-up parsers for DCG's based on the left-corner method and interpretable by PROLOG are introduced. In [8] a bottom-up parsing system making use of similar techniques has been described. They

translate the original DCG rules into a PROLOG program which incorporates subgoals governing the control of the evaluation process thus gaining efficiency as compared to ordinary DCG interpretation.
In this paper a running DCG parser with the following properties is presented:

- Grammar rules can be specified in a purely declarative way.
- Recursive rules of any type can be processed without modifications.
- There is no duplicate construction of syntactic structures.
- In case of ambiguities all applicable parse trees are constructed simultaneously.
- The dictionary can be stored in an external relational database.
- Arbitrary queries concerning the entire text base can be processed.

The paper is organized as follows. Section 2 presents the general ideas underlying the representation of Definite Clause Grammars as *LOLA* programs. In section 3 the AMOS system is presented. The paper ends with some concluding remarks in section 4.
Since the original grammar rules for old Hebrew are very complex we use a simple english example taken from [7] to explicate the essential features of the AMOS system.
The AMOS system has been developed at the Munich University of Technology in cooperation with the research group of W. Richter, Institut für Assyriologie und Hethitologie at the University of Munich. Part of the project has been funded by the "Deutsche Forschungsgemeinschaft" under contract Ba 722/3-3 "Effiziente Deduktion".

2 Grammars as Logic Programs

2.1 The Grammar Rules

LOLA is a clausal logic programming language with complex terms, (stratified) negation, and explicit existential quantification[1]. Consider the DCG rule

$$N \longrightarrow Z_1 \ldots Z_p$$

The corresponding *LOLA* rule is obtained by augmenting each (terminal or nonterminal) symbol by two attributes describing the position within the text stream:

$$\mathtt{N(I_0,I_p)\ :-\ Z_1(I_0,I_1)\ ,\ldots,\ Z_p(I_{p-1},\ I_p).}$$

See *figure 1* for an example. Extra conditions can be represented as additional subgoals in the body of the *LOLA* rule. As opposed to Prolog-related definitions of the DCG formalism we do not need to restrict the use of recursive DCG rules in any way. In particular, left-recursive rules are allowed. *Figure 2* shows sample *LOLA* rules representing a simple DCG grammar. We use the following abbreviations: s for *sentence*, np for *noun phrase*, pp for *preposition phrase*, vp for *verb phrase*, det for *determinator*, and prep for *preposition.*
(Partial) parse trees are represented as complex (Herbrand) terms which are sucessively constructed during query evaluation. In the example, the parse tree terms, e.g. s(NP_TREE,VP_TREE), occur at the additional third attribute position of rule heads and subgoals.

[1]Most of the *LOLA* systax is very close to the Prolog syntax. Variable names begin with _ or a capital letter. Constants, predicate symbols and function symbols begin with a small letter.

Input text:

John saw a man with a mirror

Positions:

```
      noun    verb    det    noun    prep    det    noun
   0       1       2      3       4       5      6       7
      John    saw     a      man     with    a      mirror
```

Figure 1: Sample input text and word positions

```
  s(X,Y,s(NP_TREE,VP_TREE))       :-  np(X,Z,NP_TREE), vp(Z,Y,VP_TREE).
  s(X,Y,s(S_TREE,PP_TREE))        :-  s(X,Z,S_TREE), pp(Z,Y,PP_TREE).
 np(X,Y,np1(N_TREE))              :-  noun(X,Y,N_TREE).
 np(X,Y,np2(D_TREE,N_TREE))       :-  det(X,Z,D_TREE), noun(Z,Y,N_TREE).
 np(X,Y,np2(NP_TREE,PP_TREE))     :-  np(X,Z,NP_TREE), pp(Z,Y,PP_TREE).
 pp(X,Y,pp(P_TREE,NP_TREE))       :-  prep(X,Z,P_TREE), np(Z,Y,NP_TREE).
 vp(X,Y,vp(V_TREE,NP_TREE))       :-  verb(X,Z,V_TREE), np(Z,Y,NP_TREE).
```

Figure 2: Rules representing simple DCG grammar

2.2 The Dictionary

The dictionary stores the terminal symbols together with their positions within the text stream that is to be analyzed. It normally depends on the particular text to be analyzed while the grammar rules themselves do not change. Furthermore, the dictionary is likely to consist of a very large number of entries (see section 3). In *LOLA* we are able to store the dictionary separate from the grammar rules as a collection of external relations where every lexical category <C> corresponds to a relation <C>. Using the SQL-database interface of the *LOLA* system [3, 6] these relations may reside on an external relational database system. At query evaluation time the appropriate (portions of) the dictionary relations are downloaded into the *LOLA* main memory database.
Position identifiers can easily be generated if the text is already separated into sentences. Languages without punctuation symbols, such as old Hebrew, require an additional pass. The position identifiers specify the string position *relative to a sentence marker.* The coding of string positions by position identifiers rather than difference lists is particularly suited to relational databases. The dictionary, i.e. the fact base of the parsing system, can be generated automatically. Normally it is the result of the morphological analysis of the input text (see section 3). As usual for DCGs, additional attributes can be used, e.g. to store casus, genus, and numerus. It occurs frequently that the morphological analysis produces ambiguous results, i.e. different classifications for the same occurrence of a word. Every such classification is stored as a separate tuple in the corresponding dictionary relation. Part of a dictionary database containing classifications of the words occurring in the sample input sentence of *figure 1* is shown in *figure 3.* Note, that the word "saw" has an ambiguous lexical classification.

```
Schema of Dictionary Database:

    noun_rel(text_id,from,to,word,casus,genus,numerus)
     det_rel(text_id,from,to,word)
    verb_rel(text_id,from,to,word,person,numerus,tense,groundform)
    prep_rel(text_id,from,to,word)

Dictionary relations:


    noun_rel = {(ch3s1, 0, 1, "John", nominative, masculinum, singular),
                (ch3s1, 3, 4, "man",  accusative, masculinum, singular),
                (ch3s1, 1, 2, "saw",  nominative, neutrum,    singular),
                 ... }
    det_rel  = {(ch3s1, 2, 3, "a"),
                (ch3s1, 5, 6, "a"),
                 ... }
    verb_rel = {(ch3s1, 1, 2, "saw", 3, singular, past, "to see"),
                 ... }
    prep_rel = {(ch3s1, 4, 5, "with"),
                 ... }
```

Figure 3: Sample dictionary database

2.3 The Grammar as a *LOLA* Program

The complete *LOLA* program representing the parser for the simple grammar of *figure 2* is shown in *figure 4*. The first part of the program contains type declarations for each predicate and function symbol occurring in the program starting with the predicate symbols which do not have a result type and followed by the function symbols. In the sample program all function symbols have the result type `tree`. Finally, the external relations and computed predicates are declared. The type system turned out to be very useful to increase the programming security. While there are reports on type systems for Prolog [9, 4] and Gödel [5], commercial PROLOG-systems use types only for non-logical constructs e.g. arithmetics.
The second part of the program contains the defining rules for the predicates, i.e. the rules representing the DCG grammar. The dictionary database is linked to the program using *LOLA*'s *database goals*[2]. The external database facility of *LOLA* automatically generates SQL-code for dictionary queries and transfers the resulting SQL-queries to the external database system. There is also an option to cluster SQL-queries as far as possible. As described above, parse trees are represented by complex terms[3]. The term representing the final complete parse tree, i.e. the term constructed by the s-rules, is graphically displayed as a tree by the user interface. To this end the computed predicate[4] `print_tree` is called when processing the `built_in` subgoal

[2]Note, that for the sake of simplicity additional attributes such as casus are eliminated by projection. The AMOS system, however, makes intensive use of additional attributes.
[3]At this point it should be mentioned that symbols can be overloaded in *LOLA* in the sense that the same symbol may denote both a predicate and a function, even of different arity, without disturbing type correctness.
[4]In the current version of *LOLA* computed predicates are implemended as Common Lisp functions. See [3]

```
$program(simple_grammar).

%------------ Declaration of Predicate Symbols ------------------------------

  ::= parse(<number>,<tree>).
  ::= s(<number>,<number>,<tree>).
  ::= np(<number>,<number>,<tree>).
  ::= pp(<number>,<number>,<tree>).
  ::= vp(<number>,<number>,<tree>).
  ::= noun(<number>,<number>,<tree>).
  ::= det(<number>,<number>,<tree>).
  ::= prep(<number>,<number>,<tree>).
  ::= verb(<number>,<number>,<tree>).

%------------ Declaration of Function Symbols ------------------------------

  <tree> ::=   noun(<word>) | det(<word>) | prep(<word>)
             | verb(<word>) | s(<tree>,<tree>)
             | np1(<tree>)  | np2(<tree>,<tree>)
             | pp(<tree>,<tree>) | vp(<tree>,<tree>).

%-------Declaration of external Relations and Computed Predicates -----------

  <$db_relation> ::= noun_rel(<text_id>,<number>,<number>,
                              <word>,<casus>,<genus>,<numerus>).
  <$db_relation> ::= verb_rel(<text_id>,<number>,<number>,
                              <word>,<person>,<numerus>,<tense>,<groundform>).
  <$db_relation> ::= det_rel(<text_id>,<number>,<number>,<word>).
  <$db_relation> ::= prep_rel(<text_id>,<number>,<number>,<word>).

  <$built_in_function([])> ::= print_tree(<tree>).


%---------------- Parse Rule and DCG Rules ----------------------------------

parse(X,Y,Tree) :- s(X,Y,Tree), $built_in([$b], print_tree(Tree)).

 s(X,Y,s(NP_TREE,VP_TREE))    :-  np(X,Z,NP_TREE), vp(Z,Y,VP_TREE).
 s(X,Y,s(S_TREE,PP_TREE))     :-  s(X,Z,S_TREE), pp(Z,Y,PP_TREE).
np(X,Y,np1(N_TREE))           :-  noun(X,Y,N_TREE).
np(X,Y,np2(D_TREE,N_TREE))    :-  det(X,Z,D_TREE), noun(Z,Y,N_TREE).
np(X,Y,np2(NP_TREE,PP_TREE))  :-  np(X,Z,NP_TREE), pp(Z,Y,PP_TREE).
pp(X,Y,pp(P_TREE,NP_TREE))    :-  prep(X,Z,P_TREE), np(Z,Y,NP_TREE).
vp(X,Y,vp(V_TREE,NP_TREE))    :-  verb(X,Z,V_TREE), np(Z,Y,NP_TREE).

%---------- Link to external Dictionary Database ----------------------------

noun(X,Y,noun(W)) :- $db($main, noun_rel(_,X,Y,W,_,_,_)).
verb(X,Y,verb(W)) :- $db($main, verb_rel(_,X,Y,W,_,_,_,_)).
det(X,Y,det(W))   :- $db($main, det_rel(_,X,Y,W)).
prep(X,Y,prep(W)) :- $db($main, prep_rel(_,X,Y,W)).
```

Figure 4: Complete *LOLA* program representing the simple grammar

```
         s
    -----|-----
   /           \
  np           vp
  |        -----|----
  |       /          \
noun    verb          np
  |      |        ----|-----
  |      |      /           \
john    saw    np            pp
             --|-        ---|--
            /    \       /     \
          det  noun   prep     np
           |    |      |     --|--
           |    |      |    /     \
           a   man    with det   noun
                            |     |
                            |     |
                            a   mirror
```

```
                  s
          --------|--------
          /                \
        s                 pp
    ----|----            --|--
   /         \          /     \
  np          vp     prep    np
  |         ---|-       |    --|--
  |        /     \      |   /     \
noun    verb     np with  det   noun
  |       |      -|-       |      |
  |       |     /   \      |      |
john     saw  det  noun    a    mirror
               |    |
               |    |
               a    man
```

Figure 5: Different parse trees

of the parse rule. The query `:- parse(0,X,Tree).` results in the construction and display of all possible parse trees for complete sentences[5]. The first argument of `parse` represents the begin position, the second the end position of the sentence parsed. The third argument contains the parse tree term. In our example the answer relation consists of three answer tuples, two of them with end position 7 and one with end position 4. The display of the former two parse trees is visualized in *figure 5.*
The reader may have noticed, that we use different function symbols as constructors of the parse trees while the parse trees shown in *figure 5* do not make this distinction. To preserve type correctness we have to distinguish the noun phrase constructors of arity 1 (`np1`) from those of arity 2 (`np2`). On the other hand, we want to display the parse trees with maximum convenience for linguists. The `print_tree` function performs the appropriate processing of the computed parse trees.
Instead of implementing special purpose display routines such as `print_tree` the explanation facility built-in to the *LOLA* system [14, 15], can be used to display the parse trees together with information how the particular trees have been derived, i.e. which rules have been applied etc..

3 The AMOS Parsing System

The AMOS system for the morpho-syntactical analysis of old hebrew text has been implemented applying the techniques described above. The grammar for old Hebrew [12] has been formalized as DCG representation and written as a *LOLA* program [13]. The dictionary relations are

for more details on computed predicates.

[5]For the AMOS system it is important to consider all sentences and not only those of maximum length since punctuation is lacking in old Hebrew.

generated by the morphological analysis system SALOMO for old Hebrew [1] and can be stored in an external relational database.
The sample AMOS rules[6]

```
attP(Sentence,X,Y,advP1(N_TREE,ADJ_TREE),Casus,Genus,Numerus) :-
        noun(Sentence,X,Z,N_TREE,Casus,Genus,Numerus),
        adj(Sentence,Z,Y,ADJ_TREE,Casus,Genus,Numerus).

attP(Sentence,X,Y,advP2(A_TREE,ADJ_TREE),Casus,Genus,Numerus) :-
        attP(Sentence,X,Z,A_TREE,Casus,Genus,Numerus),
        adj(Sentence,Z,Y,ADJ_TREE,Casus,Genus,Numerus).
```

represent the grammar rule for the *attribute-phrase (attP)*: *An attribute-phrase in old Hebrew is a noun, followed by an adjective, or an attribute-phrase followed by an adjective.* In addition, the rules specify that congruence of casus, genus and numerus is required. The following (simplified) AMOS rules show that even nonlinear recursion is necessary. The *apposition-phrase (appP)* defines compound nouns and compound noun-phrases:

```
appP(Sentence,X,Y,appP(N_TREE1,N_TREE2))  :-
        noun(Sentence,X,Z,N_TREE1,absolutus, _, _),
        noun(Sentence,Z,Y,N_TREE2,_ , _, _).

appP(Sentence,X,Y,appP(N_TREE,A_TREE))  :-
        noun(Sentence,X,Z,N_TREE,absolutus, _, _),
        appP(Sentence,Z,Y,A_TREE).

appP(Sentence,X,Y,appP(A_TREE1,A_TREE2))  :-
        appP(Sentence,X,Z,A_TREE1),
        appP(Sentence,Z,Y,A_TREE2).
```

The AMOS program contains several more appP-rules which are more complicated and selective. A section of the dependency graph of the AMOS program is shown in *figure 6*. The very beginning of the book of the Ecclesiasts is an example of an apposition-phrase to which the above nonlinear recursive rule is applicable. A literal translation results in:

```
words Kohelet son David king   of   Jerusalem
  |      |     |    |    |     |         |
noun   noun  noun  noun noun preposition noun
```

This can be parsed in several ways (using additional morphem attributes not mentioned here) corresponding to the following interpretations:

- *words [of] Kohelet [who was a] son [of] David [and David was] king of Jerusalem,*
- *words [of] Kohelet [who was a] son [of] David [and every son of David was] king of Jerusalem,*
- *words [of] Kohelet [who was a] son [of] David [and who (Kohelet) was] king of Jerusalem,*
- *words [of] Kohelet [who was a] son [of] David [and words of the] king of Jerusalem.*

AMOS derives a parse tree for each of the above interpretations. Linguists are often very interested in finding such (syntactical) ambiguities.

[6]The real AMOS system contains more attP and appP rules than shown here. Note, that the schema of logic predicates and base relations is extended as compared to the simple examples shown in section 2.

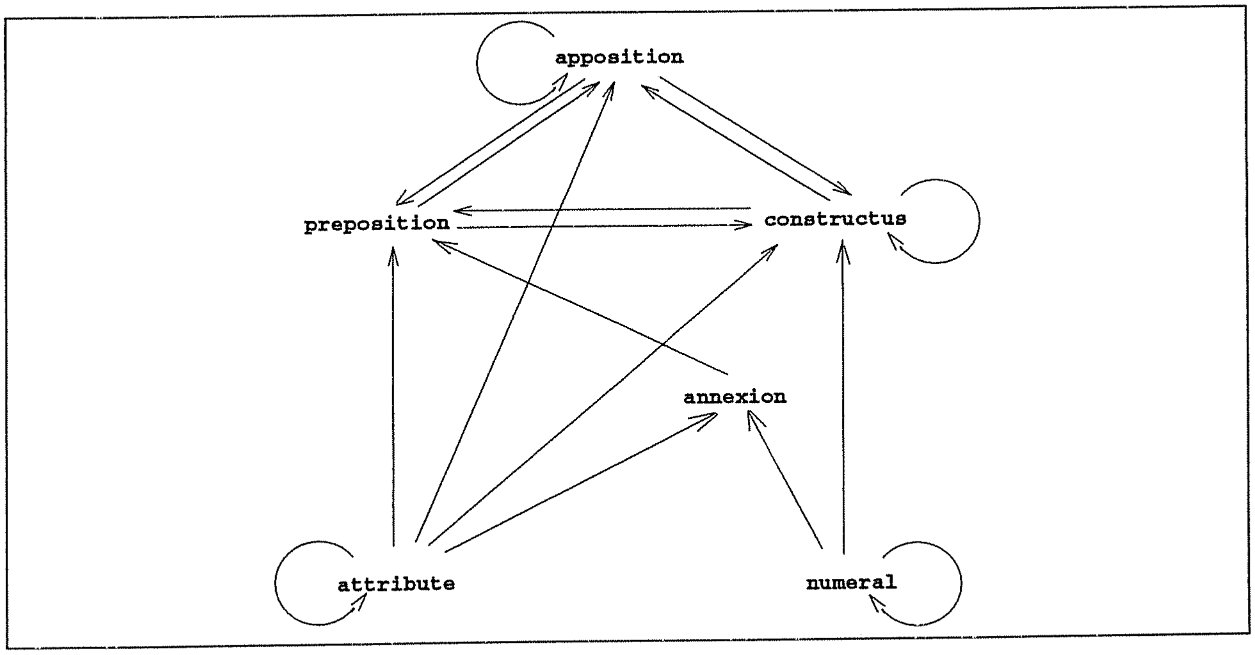


Figure 6: Section of the dependency graph of the AMOS program

The characteristics of the AMOS system are

- Large volumes of text like the *Old Testament*;
- Ambiguities in the dictionary due to words not uniquely classifiable at the morphological level;
- Ambiguous parse trees as derived using the DCG rules.

The dictionary database consists of 25 relations with a total of 117 attributes. The extension of the database relations depends on the text which is to be analyzed and may be rather large thus preventing the downloading of the entire dictionary. However, we can still benefit from the advantages of deductive databases. Since AMOS adopts the set-at-a-time evaluation strategy of *LOLA* there is no need to parse one sentence after the other as in Prolog-based systems. We found that processing an entire paragraph or chapter gives the best performance results. On the other hand, database support is needed to store the input text as well as the computed answer relations. The AMOS dictionary for one chapter of the *Genesis* contains 167kBytes with 4282 tuples. The entire *Genesis* has already 7.250 MBytes with 185 897 tuples, and the *Old Testament* has 108.750 MBytes with 2.7 Mio. tuples.
The complete morpho-syntactical parser for old Hebrew is represented by about 200 *LOLA* rules. The program contains 4 linearly recursive, 13 mutually recursive, and 1 nonlinearly recursive (quadratically recursive) predicate symbols.
An overview of the AMOS system architecture is shown in *figure 7*. Frequently used queries are precompiled and can be invoked by the user. In addition, the system allows ad-hoc queries. The system is running on UNIX workstations with Allegro Common Lisp, Lucid Common Lisp, and AKCL.

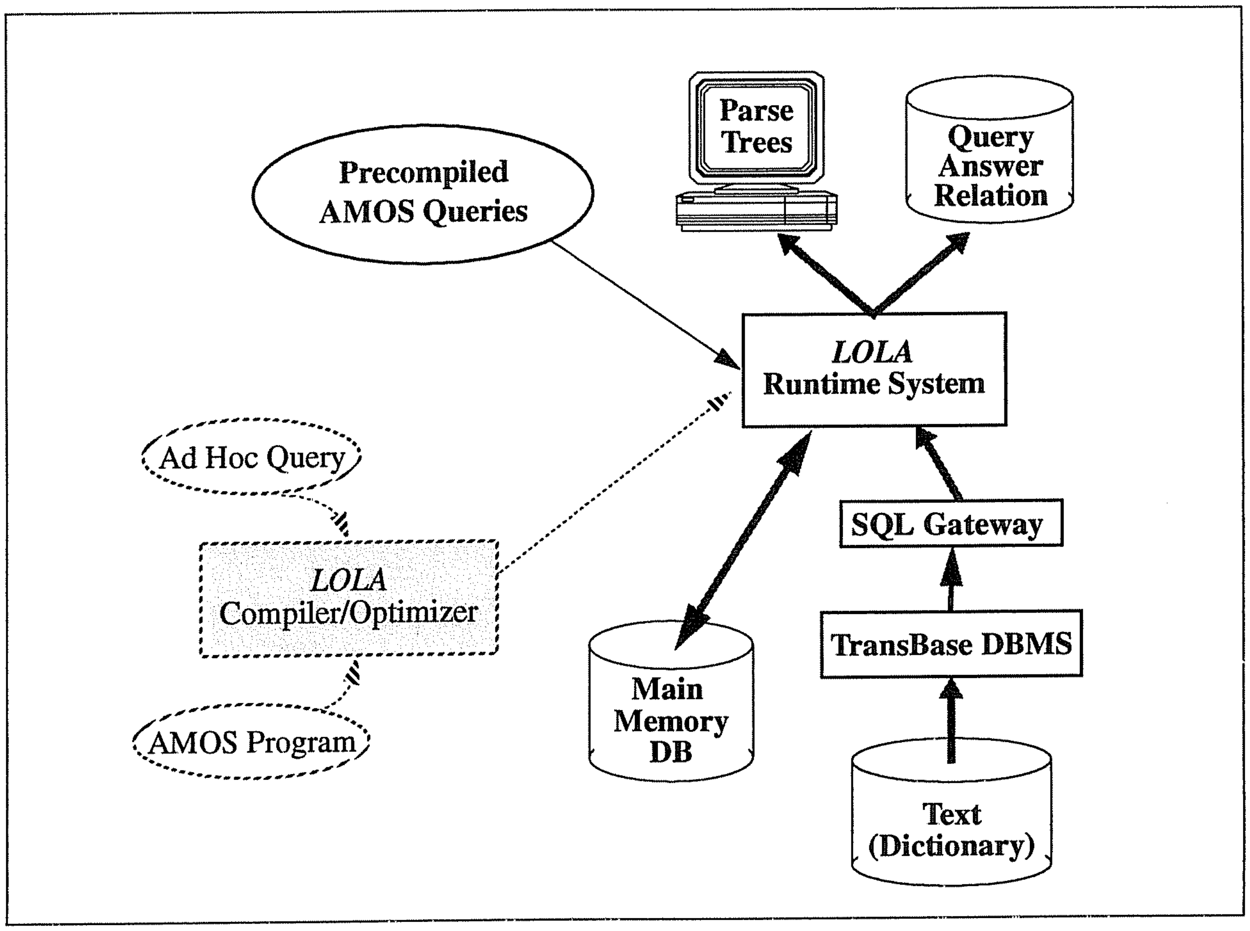


Figure 7: Overview of the AMOS system architecture

4 Conclusion

We have described the AMOS parsing system for old hebrew text. The system has been implemented using the *LOLA* system. Only the predicate causing side-effects, i.e. `print_tree`, had to be encoded in the host language. There are considerable advantages as compared to Prolog based systems, e.g. the indepence of rule order and the ability to handle any type of recursion directly.

Acknowledgement

We would like to thank W. Richter and W. Eckardt, Seminar of Hebraistic and Ugaristic at the University of Munich (LMU), for supporting the formalization of the grammar of old Hebrew.

References

[1] W. Eckardt. *Computergestützte Analyse althebräischer Texte: Algorithmische Erkennung der Morphologie*, volume 29 of *Arbeiten zu Text und Sprache im Alten Testament.* EOS Verlag, St. Ottilien, Germany, 1987. (In German).

[2] B. Freitag, H. Schütz, and G. Specht. *LOLA* — a logic language for deductive databases and its implementation. In *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA '91)*, pages 216 - 225, Tokyo, 1991.

[3] B. Freitag, H. Schütz, G. Specht, R. Bayer, and U. Güntzer. *LOLA* - a deductive database system with integrated SQL-database access. Technical report, Technische Universität München, 1993.

[4] M. Hanus. Horn clause programs with polymorphic types: Semantics and resolution. In *Proc. TAPSOFT'89*, volume 352 of *LNCS*, pages 225 - 240, Berlin, 1989. Springer-Verlag.

[5] P. M. Hill and J. W. Lloyd. The Gödel report. Technical Report TR-91-02, University of Bristol, Dep. of Computer Science, 1991.

[6] J. Kempe, T. Lenz, B. Freitag, H. Schütz, and G. Specht. CL/TB - an Allegro Common Lisp programming interface for TransBase. *ACM SIGPLAN Notices*, 26(8):60 - 69, 1991.

[7] B. Lang. Datalog automata. In *Proc. 3rd Intl. Conf. on Data and Knowledge Bases: Improving Usability and Responsiveness*, pages 389–404, Jerusalem, 1988.

[8] Y. Matsumoto and R. Sugimura. A parsing system based on logic programming. In *Proc. Intl. Joint Conf. on Artificial Intellingence (IJCAI'87)*, pages 671 - 674, 1987.

[9] A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[10] F. Pereira and S. Shieber. Prolog and natural-language analysis. Technical report, Stanford University, 1987. CSLI Lecture Notes Nr.10.

[11] F. Pereira and D. Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.

[12] W. Richter. *Grundlagen einer althebräischen Grammatik: B. Die Beschreibungsebenen, II. Die Wortfügung (Morphosyntax)*, volume 10 of *Arbeiten zu Text und Sprache im Alten Testament.* EOS Verlag, St. Ottilien, Germany, 1979. (In German).

[13] G. Specht. *Wissensbasierte Analyse althebräischer Morphosyntax: Das Expertensystem AMOS*, volume 35 of *Arbeiten zu Text und Sprache im Alten Testament.* EOS Verlag, St. Ottilien, Germany, 1990. (In German).

[14] G. Specht. *Source-to-Source Transformationen zur Erklärung des Programmverhaltens bei deduktiven Datenbanken.* PhD thesis, Technische Universität München, 1992. volume 42 of DISKI, infix Verlag, St. Augustin, 1993 (In German).

[15] G. Specht. Generating explanation trees even for negations in deductive database systems. In *Proc. 5th Workshop on Logic Programming Environments, in conjunction with Int. Logic Programming Symposium (ILPS '93)*, Vancouver, 1993.

A New User's Impressions on $\mathcal{LDL}$++ and $\mathcal{CORAL}$

Ping-Yu Hsu and *Carlo Zaniolo*
Computer Science Department
University of California
Los Angeles, CA 90024

1 Introduction

The genesis of this submission is the graduate class on Deductive Databases offered at UCLA in the Spring of 1993. The first author learned $\mathcal{LDL}$++ and implemented the examples described in this paper and other applications as class assignments; then he proceeded with learning the Coral system, re-developing these applications and additional ones on such a system, and submitting a comparative study as a term-paper for the course. The second author suggested further experiments and made editorial improvements aimed at communicating the valuable know-how thus acquired on the relative strengths and limitations of the two systems. The technical impressions, however, remain those of the first author—a new user to both systems.

A first objective of this paper is helping new users, and a second one is to provide some feedback to deductive database researchers and system designers. In this perspective, rather than pursuing a direct comparison between the two systems, the paper compares the factual views derived from our experience with the systems with the a-priori expectations derived from reading various papers and manuals authored by the designers.

According to its designers, $\mathcal{LDL}$++'s main goal (and improvement with respect to the old $\mathcal{LDL}$ system) was that of making the system more friendly for new users and naive users. Thus $\mathcal{LDL}$++ 's design rationale lead to a minimalist's " keep-it-simple approach": the language was kept simple and declarative [Zan2]. Moreover, great efforts were made to provide seamless interfaces to SQL databases and C++, inasmuch as most real-world applications were expected to be in conjunction with pre-existing databases and software packages [Zan2].

According to its designers, Coral's goal is that of providing a wide range of evaluation and optimization strategies to ensure better performance and a richer programming environment [RSS 92]. To that end, Coral allows the programmer to control the optimization strategies used, and to exploit the operational semantics of programs: Coral supports an imperative module in addition to interfaces to C++.

Therefore, the design goals of the two systems are very different, and our experience confirmed that both systems have largely achieved their original design goals. However, this positive conclusion holds only as a first approximation, since in both systems there is room for improvement.

By revealing these limitations, this study will help a hopefully growing throng of new users to calibrate their expectations, and will encourage the designers and implementors to improve their systems and overcome current limitations in the next release.

2 Using the Systems

2.1 Ease of Learning

Enticing new users and less sophisticated programmers is the stated goal of $\mathcal{LDL}$++ [Zan2]. In fact, our experience confirms that $\mathcal{LDL}$++ seems easier to learn. In $\mathcal{LDL}$++, the user only needs to become familiar with the logical rules: there is only one meaning for each program and every part of the logic is in the program. On the other hand, Coral supports a rich set of options, such such as pipeline, no_rewriting, select, aggregate selection, etc., which modify the meaning of programs in order to make them more efficient. Our experience indicate that these can complicate the life of first-time users and more naive programmers, who have to learn both the "normal" meaning of a program and the variations caused by annotations. (Occasionally, to really understand the meaning of annotations a user must become familiar with the ways in which Coral evaluates and optimizes programs.) The presence of an imperative module, and the possibility of having delete goals in rules, can also add to the complexities faced by a first-time user. On the other hand, it should be noted that only few annotations are actually needed to control the meaning of programs (as opposed to controlling its performance). Thus, novices should be encouraged to ignore annotations and concentrate on mastering the programming paradigm under the basic default options provided by the system. This more didactic approach deserves to be fully explored in the future, since it appears capable of overcoming most problems previously discussed.

While analysis confirmed that $\mathcal{LDL}$++ was attuned toward first-time users and naive programmers, this conclusion must be qualified by several caveats and additional considerations. Much of the effort of learning how to program in $\mathcal{LDL}$++ was not specific to the system but, rather, had to do with the problem of mastering the deductive programming paradigm—e.g., safe rules, function symbols and recursion. Students, such as the first author, who had a previous experience with Prolog, were favored in certain respects, but they also seem to have initially a harder time with certain concepts such as the bottom-up execution of recursive predicates. The task of learning the deductive programming style represents a large percentage of the overall investment required from a new user and is basically independent of the system. For instance, when learning $\mathcal{LDL}$++, the first author benefited from a two-hour introduction by the instructor, and used his previous Prolog experience. Yet, to master the deductive programming paradigm, he had to spend several hours on the system. New users to Coral should also be prepared to dedicate additional time to learn how to make effective use of its non-declarative constructs.

Finally, the current versions of $\mathcal{LDL}$++ and Coral have inadequate debugging and tracing facilities. In both systems, the output generated in debugging mode is obscure, and almost impossible for a new user to understand—hard for experienced users as well.

2.2 Functionality

After the initial learning phase, users reach the stage in which they are reasonably familiar with a system and know how to design and develop applications with it. At this stage, users' satisfaction depends on several such as the level of confidence and productivity they have achieved. In the following sections, we evaluate various characteristics that are most critical in this respect, such as functionality, completeness, reliability and development turn-around time.

Both $\mathcal{LDL}$++ and Coral provide the necessary functionality to write applications in the rule-based programming paradigm. The basic rules in the two languages are in fact very similar, but there are differences in the data types and built-ins they support. These differences can impact the the usability of a system and the performance of applications. For instance, the *if-then-else* statement of $\mathcal{LDL}$++ can be easily replaced by a pair of rules containing negation, but it is hardly redundant from a practical viewpoint, inasmuch as it makes the program more expressive and efficient.

Coral supports a rich set of built-functions, such as printf, which allows a user to print some output in the screen, and display_timer(), which allows a user to get the profile of a program running time. $\mathcal{LDL}$++ is lacking some of these built-ins; in particular, it only supports a limited form of printf statement (and current documentation on this topic leaves a lot to be desired). This limitation is particularly hard on new users who need to print intermediate results on screen for debugging purposes.

Both systems support sets and lists, but Coral also supports multisets. However, our release of Coral seems to be missing some important functions, including operators such as union or append, that are needed in this context. Also, variables are not allowed in sets (i.e., between { and }).

Coral supports a rich collection of ready-to-use set-aggregates, but the set-aggregate mechanism in $\mathcal{LDL}$++ is more powerful inasmuch as it allows users to define his/her own aggregation functions. Experienced users are likely to be pleased with the power that this extensibility entails (at little cost for them, inasmuch as they will quickly re-use and modify set-aggregate definitions used in the past). New users, however, are likely to find the task of writing the basic aggregates too taxing. An ideal system should, perhaps, provide a predefined library of set-aggregates along with user-programmability.

2.3 Development Environment

In Coral, programs can be separated into several modules and only the parts that are updated have to be re-compiled. Modules are described in several $\mathcal{LDL}$++ papers, but are not implemented yet; in the current $\mathcal{LDL}$++, the whole program has to be re-compiled after each change. $\mathcal{LDL}$++ achieves a reasonable speed of compilation, but, in Coral, the same program will normally compile faster; moreover Coral's modules further expedite the turn-around on compilation.

Neither system is mature and stable enough to prevent nasty surprises for a user, particularly when an error occurs at run time. $\mathcal{LDL}$++ seems more stable since we only experienced strange behavior when the programs had bugs, and the system would seldom aborted even in the presence of bugs. (With Coral, aborts are more frequent, and we had a correct program that would ran after it was split over several modules, but not in the original form.)

2.4 Choice

Our study did not dwell into areas, such as non-monotonic constructs in negation, which represent open areas of current research for which the systems provide very different solutions [Zan2]. However, we pursued a comparison of the choice constructs, which were similar in the two systems, and were used frequently in our applications.

$\mathcal{LDL}$++ provides choice as a built-in predicate, while the approach of Coral is to use annotation [RSS 92]. The scope of a choice function in $\mathcal{LDL}$++ is the body of the rule containing the choice goal. The scope of annotation in Coral is the set of all rules defining a predicate (i.e., a procedure in logic programming parlance). In terms of expressive power, the two approaches are equivalent, since it is easy to show that by the introduction of additional rules or predicates one approach can simulate the other. For specific applications however, one approach may lead to terser programs than the other.

For instance, finding the shortest distance between pairs of nodes of a graph, where each edge counts as a unit of cost, can be expressed in Coral as follows (in our Coral system we had to simulate choice with min).

Example 1 (Choosing using min in Coral)

```
dist(X,Y, 1)    ←  edges(X,Y,_).
dist(X,Y, Dist) ← dist(Z,Y,Dist1), edges(X,Z,_),
                   Dist = Dist1 + 1.
@aggregate_selection dist(X,Y,Dist) (X,Y) min(Dist).
```

Here, no two arcs in `dist`, can have the same end-node, independent of the rule that produced them, since Coral imposes a constraint over the whole predicate. This example illustrates an effective use of annotations in Coral—and the fact that "annotations" play a critical role in defining the the meaning of a Coral program.

In $\mathcal{LDL}$++ instead, the functional dependency constraint declared by choice holds only for the atoms generated by the rule containing choice. Thus, the constraint does not "see" values generated by other rules, and in our particular case, the Y's added by the first rule. Thus an additional goal $\neg edges(X, Y, \_)$ must be added to avoid edges looping back to those old nodes.

Example 2 ($\mathcal{LDL}$++ with Choice)

```
dist(X,Y, 1)    ←  edges(X,Y,_).
dist(X,Y, Dist) ←  dist(Z,Y,Dist1), edges(X,Z,_),
                   ¬ edges(X, Y, _ ) , Dist = Dist1 + 1,
                   choice((X,Y),(Dist)).
```

3 Performance

It is well-known that any database vendor can produce benchmarks where a given system outperforms the others. This ended-up being also the case for the small set of applications, which were given as a class assignment with sole intent of having the students to learn concepts, such as stratification and constant pushing, by writing a toy compiler for a Datalog-like language. The code for three such applications, called *notstratify*, *constantpush* and *max_depth*, are given in the Appendix. The program *notstratify* tests the handling of lef/right recursive programs by the systems; the program *constantpush* stresses the ability of the two system of caching intermediate results to compute faster, while the program *max_depth* evaluates the uses of choice.

The results of running these programs are shown in Table 1, Table 2 and Table 3. These benchmarks were measured on a dedicated Sun Microsystems workstations, with a memory capacity sufficiently large (32Mbytes) as to exclude that page faults would have a major impact on the performance of the two systems. The versions of the two systems we used were those current in September 1993.

The Coral designers have put a significant effort[1] into optimizing the performance of programs [RSS 92]. This effort has paid up handsomely, and, for simple programs, Coral has a performance advantage. For programs consisting of few rules, each rule containing only one or two joins on database relations, we found the execution time of Coral to be in the order of 40% to 65% below that of $\mathcal{LDL}$++. (These execution times are comparable to those of the old $\mathcal{LDL}$ system, but without its compilation overhead.) However, we found that this advantage can diminish, or even disappear, in some complex programs, such as those that use linear recursion and larger modules.

3.1 Discussion

Tables 1 and 2 provide a good illustration of the fact that comparing performance is never a simple task. A Coral program can be written either in several modules or in one module. In Table 1, the execution times improve once the modules are merged, while in Table 2 they grow. A possible explanation of this behavior is that we are seeing the effect of some preparation time (e.g., setting up the supplementary magic tables). This effect becomes more visible in in Table 2, since the overall execution times are reduced dramatically. In

[1]This has concentrated on high-level optimization. Lower level, WAM-like optimization can yield significant additional improvements, both for Coral and $\mathcal{LDL}$++.

Table 1 instead, the overall execution dominates the preparation phase and times improves as the modules are merged together.

Thus, Coral users should be aware that, at times, it is better to optimize Coral programs by breaking them into modules, even if there is no optimization between different modules. For instance, *notstratify* in Coral is 16% faster than that in $\mathcal{LDL}$++ when the program is written in the same module and about the same speed when the program is written in different modules.

	LDL++	Coral(five modules)	Coral(same module)
notstratify	62.66	61.26	53.78
max_depth	2222.49	302.20	105.70
constantpush	27.21	76.20	58.33

Table 1: Intermediate predicates are not stored to database

An important lesson for the users of both systems is on the importance of storing intermediate data. Both $\mathcal{LDL}$++ and Coral offer this ability, but in very different ways. In $\mathcal{LDL}$++, a predicate with prefix '+' signifies the system to store the predicate in database, whereas, in Coral, a separate imperative module is necessary to store a predicate into database. In our programs, predicates of dep_atom/4 and dep_rule/5 are derived predicates and are referenced by many rules. Table 1 shows the times obtained without storing dep_atom/4 and dep_rule/5 in the database. Table 2 shows that the times for the same programs with dep_atom/4 and dep_rule/5 stored in database can be a order of magnitude smaller. Therefore, the users should be aware that storing intermediate data can save a lot of time in both $\mathcal{LDL}$++ and Coral [2].

Tables 1, 2 and 3 contain several other points that should be of interests to the designers and users of the two systems. A first concern pertains to the speed with which the two systems implement recursive predicates. Several examples of lef-linear recursion occurred in our examples. The comparative times for *notstratify* were typical of these examples. Thus, it appears that for this kind of recursion, Coral loses the advantage that it normally has for non-recursive rules. Thus, we concluded that $\mathcal{LDL}$++ must have implemented a more efficient approach to left-linear recursion.

Consider now the *constantpush* entry in Table 1. In this example, $\mathcal{LDL}$++ has a performance advantage which is well beyond what can be explained with a better scheme for left-linear recursion. In our view, this instead reflects the capability of a system to reduce duplicate work, as when there is caching of

[2]The time spent in storing the facts in the database was a negligible fraction of the overall time.

old results for future usage. Explicit caching is available in Coral's through the "save_module" annotation; but in addition, both systems incorporate some degree of implicit caching in their executions. In $\mathcal{LDL}$++ this is accomplished through a client-server architecture, whereby several calling goals (the clients) can request a next-tuple from the server that, e.g., returns a lazily-computed new tuple of a recursive predicate. This strategy appears to work well in the example at hand. The kernel of *constantpush* is *reachable/2*. Under our example, *reachable/2* is executed eight times and five of them are repetitions of previous requests. In $\mathcal{LDL}$++, a single query "reachable(7,X)" takes 16.81 seconds. Encapsulated in constantpush, the eight calls to reachable/2 take less than 27.21 seconds. On the other hand, a single query "?reachable(7,X)" takes about 9 seconds in Coral and the eight calls on reachable/2 in constantpush take about 76 seconds when programs are written in several models.

In many cases, however, Coral's execution implicitly achieves some of the benefits of caching. For instance, we know that *conspush* is executed twice in *constantpush*. Hence, the execution time should be at least 9 times 8 times 2, which is 144. In fact, from Table 1, the execution time is 76 seconds, which is only half of the expected time. Thus, when the second conspush is requested Coral only takes 0.11 additional seconds. In other situations, the savings is less than what it is achievable. Thus, five of the eight executions of reachable/2 are repetitions. From Table 1, however, merging programs only improve Coral's performance from 76.20 to 58.33 seconds. Here, Coral waits 28 seconds before returning the first answer to the query reachable/2 (a time longer than the total execution time in $\mathcal{LDL}$++). Since a duplicate query is answered later at almost no cost, we ventured to guess that this behavior is actually the result of the supplementary-magic execution strategy Coral uses.

	LDL++	Coral(five modules)	Coral(same module)
notstratify	7.77	6.39	17.59
max_depth	11.90	5.15	8.64
constantpush	0.98	5.70	16.07

Table 2: Frequently referenced predicates are stored in database

We also found several areas which require further attention by the $\mathcal{LDL}$++ designers and implementors. The program *max_depth* combines choice and linear recursion. From Table 1, we see that $\mathcal{LDL}$++ is significantly slower than Coral, albeit the difference is reduced when intermediate predicates are stored—Table 2. (The engine of *max_depth* is *dist/3* which was invoked 462 times in this experiment.) Given $\mathcal{LDL}$++'s good performance on linear recursion, we cannot but blame $\mathcal{LDL}$++'s `choice` as the culprit (although we have not decided yet whether this is merely an optimization issue, or semantic differences with respect to Coral constructs also play a role).

Another weak point of $\mathcal{LDL}$++ was non-linear recursion. The 'naive' definition that computes Fibonacci number by a quadratic rule, would run much faster in Coral than in $\mathcal{LDL}$++, to an extent that suggests that Coral is using a better compilation method.

	LDL++	Coral(five modules)	Coral(same module)
notstratify	7.41	6.30	20.08
max_depth	36.22	11.60	19.65
constantpush	2.05	16.34	73.93

Table 3: Doubling the data with frequently referenced predicates stored in DB

While this comparative study has been of very limited scope, it has has nevertheless identified areas of opportunity for improvement in both systems.

Several other tests were performed in addition to those presented in the tables. While these experiments confirmed our tentative conclusions discussed above, they pointed out several other areas that deserve further investigations and a second look by the designers. For instance, Table 3 shows the same programs as Table 2 once the size of the database is doubled. The execution time of `notstratify` does not change much (as expected since the time needed to run its main procedure `path` does not increase much). The times for the other two programs more than double, in a fashion that seems to indicate that there are several factors involved, which deserve further investigation.

4 Conclusion

This paper represents only a modest first step in the right direction, but, in our view, a useful one. By comparing a new user's experience with two of the leading deductive database systems we have derived know-how that can help both the users and the designers.

In particular, new users of the systems should have derived a better understanding about the strengths of the two systems and how to use them to their advantage (e.g., in the areas of modularization and caching).

In terms of the relative performance of these systems, our results indicate, that there are several areas where either system shows strengths or weaknesses with respect to the other.

The designers are thus given the opportunities of understanding the sources of such differences and to improve their systems.

The applications of the lessons learned in this study will foster the progress of deductive database technology and its deployment in commercial applications.

References

[RSS 92] Raghu Ramakrishnan, Divesh Srivastava and S. Sudarshan. CORAL-Control, Relations and Logic. *Proceedings of the 18th VLDB Conference*, 1992.

[SHAM 1989] Shamim Naqvi and Shalom Tsur, *A Logical Language fro Data and Knowledge Bases*, Computer Science Press, 1989.

[SRIV 92] Davesh Srivastava, S. Sudarshan and Raghu Ramakrishnan, The Manual of Coral, January, 1993.

[Zan2] Zaniolo, C., *Intelligent Databases: Old Challenges and New Opportunities*, Journal of Intelligent Information Systems, 1, 271-292 (1992).

Acknowledgments The authors thank R. Ramakrishan and the referees for several suggested improvements.

A Programs

A.1 The Program Max_Depth

The basic program consists of that described in Section 2.4, and the following rules:

```
max_depth(Dist)←distance(D), max(D, Dist).
distance(⟨D⟩)← dep_atom(X,_,_,_), dep_atom(Y,_,_,_),
               X ~= Y, dist(X,Y,D).
```

A.2 The Program Notstratify

The main purpose of notstratify in this paper is to compare the strength of $\mathcal{LDL}$++ and Coral in handling linear recursive program. The most important and time consuming predicate in notstratify is path/4. The predicate of path/4 is defined as following in $\mathcal{LDL}$++

```
path(Cur, DST, {DST}, {Rule})←      edges(Cur, DST, Rule).
path(Cur, DST, Visted, NSSTR)←      path(Mid, DST, Visted1, NSSTR1),
                                    edges(Cur, Mid, Rule),
                                    % Mid should not have been traversed
                                    ~member(Mid, Visted1),
                                    union({Mid}, Visted1, Visted),
                                    union(NSSTR1, {Rule}, NSSTR).
```

Edges/3 can be viewed as facts in this case and member/2 and union/3 are system defined functions. Hence, path4/ is a linear recursive predicate.

Path/4 of Coral version is as follows.

```
path(Cur, DST , Ds, Rs)←          edges(Cur, DST, Rule),
                                  add(DST,{},Ds), add(Rule,{},Rs).
path(Cur, DST, Visted, NSSTR)←    path(Mid, DST, Visted1, NSST R1),
                                  edges(Cur, Mid, Rule),
                                  % Mid should not have been traversed
                                  not member(Mid, Visted1),
                                  add(Mid, Visted1, Visted),
                                  add(Rule, NSSTR1, NSSTR).
```

Edge/3 can be treated as facts, member/2 is a system defined function and add/3 is a user defined, non-recursive function which added an element to a set.

The reason for the change is that set union is not implemented and a variable is not allowed to appear between { and } in Coral. While differences between union/3 of $\mathcal{LDL}$++ and add/3 of Coral are also a factor in determining the running times, we concluded that this was not the major one.

A.3 The Program Constantpush

Constantpush also employed linear recursion. However, in this case, we emphasizes more on the ability of being able to cache intermediate result than compute the result fast. The most important predicate in this example is reachable/2, which is a linear recursive program. In our test example, reachable/2 is called eight times with three sets of arguments. That means five out of the eight executions are repetition. If the system comes with a good cache system, then the execution time can be reduced dramatically.

The program of reachable/2 looks about the same in both versions. In $\mathcal{LDL}$++ we have:

```
% ⟨S⟩ is the set of nodes that can be reached by P
reachable(P,⟨S⟩)← reach(P,S).
% reachable/2 never fails
reachable(P,{})← ~reach(P,S).
% check whether there is an path between P,S
reach(P,S)← edges(P,S,_).
reach(P,S)← reach(Mid,S), edges(P,Mid,_).
```

The point here is not how much time the system spent to compute reachable/2, but how much time the system saves by using cache.

In Coral, the statement

```
conspush(Pn, Arity, Argtemp, ⟨Rule⟩,⟨Atoms⟩)←
          cp(Pn, Arity, Argtemp,X,T,Z,Rule,Atoms).
```

is illegal. Thus, the program was changed to

```
conspush_rule(Pn, Arity, Argtemp, makeset(⟨Rule⟩),_)←
               cp(Pn, Arity, Argtemp,X,T,Z,Rule,Atoms).
conspush_atom(Pn, Arity, Argtemp, _,makeset(⟨Atoms⟩))←
               cp(Pn, Arity, Argtemp,X,T,Z,Rule,Atoms).
```

Reachable/2 is called by cp/7. In Coral, cp/7 is called twice with the same input argument. That means reachable/2 may be invoked sixteen times, instead of eight times. This scenario could in theory damage Coral's performance a lot. To our surprise, however, we found that this repetition did not hurt the actual performance much.

LogicBase: A System Prototype for Deductive Query Evaluation*

Jiawei Han, Ling Liu and Zhaohui Xie
School of Computing Science
Simon Fraser University
Burnaby, B.C., Canada V5A 1S6
{han, lingliu, zhaohui}@cs.sfu.ca

Abstract

A prototyped deductive database system, LogicBase, is being developed in Simon Fraser University as a testbed for efficient query evaluation in deductive database systems. This paper introduces the LogicBase project: its design principles, query evaluation methods, implementation considerations and some testing results. A major feature of the LogicBase system is its chain-based query evaluation method which is unique in the family of deductive database systems. The method is motived by the following observations: (1) most popularly studied recursions can be compiled into highly regular chain or pseudo-chain programs; (2) the compilation may capture the bindings which could be difficult to be captured otherwise; and (3) because of the regularity, simplicity and exactness of compiled chain programs, efficient query evaluation can be performed by exploration of the available query constraints, integrity constraints, recursion structures, and other features of the programs. Interesting chain-based evaluation techniques, such as chain-following, chain-split, constraint pushing, partial evaluation, etc., have been developed in the project. In this paper, we introduce the chain-based query evaluation method, briefly present the design and implementation of the LogicBase project, compare the chain-based query evaluation method with other methods, and outline the future research and development plan of the project.

1 Introduction

As an important extension to relational database approach, research on deductive database represents a promising direction towards declarative query processing, high-level database

*This work was supported in part by the *Natural Sciences and Engineering Research Council of Canada* under the grant OPG-3723 and a research grant from the *Centre for Systems Science of Simon Fraser University*.

programming, and the integration of logic programming and relational database technology [23]. Deductive database technology has wide applications in business data management, engineering databases, spatial databases, and knowledge-base systems [6, 26]. With the maturity of deductive database research, many deductive database systems or prototypes, such as LDL [6], Glue-NAIL! [18], CORAL [20], EKS-V1 [30], ADITI [29], etc. have been developed and reported in recent years.

Although many deductive database systems adopt the syntax of logic programming languages, the interface for a deductive database system can be SQL-like, icon-based, etc. The system can be constructed based on an extended relational or entity-relationship model or a deductive and object-oriented (DOOD) model and be supported by an object-oriented backend [28]. A database system is deductive if it supports high-level views by sophisticated deduction rules and a declarative query interface.

Since a deductive data language extends a relational query language to at least Horn clause logic, efficient evaluation of recursions in deductive databases has been an important issue in deductive database research [2, 27]. As many researchers [21, 2, 6, 18, 20, 29] have noted that Prolog implementations are inappropriate for deductive database applications due to its order-dependent and tuple-at-a-time evaluation, repeated computation of subgoals, possible infinite recursive looping, and its difficulty at judging the completeness of the search.

Many techniques have been proposed and studied for efficient query evaluation in deductive databases [16], which results in two influential classes of deductive query evaluation methods: (1) top-down evaluation, represented by the query/subquery approach [30], and (2) bottom-up evaluation, represented by magic sets computation and semi-naive evaluation [2, 6, 18, 20, 29]. These methods explore set-oriented evaluation, focus of the search on query relevant facts, with freedom of looping and easy termination testing, and have achieved impressive results. However, because a recursion is more or less treated as a black box by these methods without a detailed analysis of its particular structure, it is difficult to capture the regularities of a particular recursion and maximally utilize the information about constraints and recursion structures in query evaluation.

The LogicBase project adopts a different approach in deductive query evaluation, which relies on query-independent compilation and chain-based query evaluation. The former (query-independent compilation) [13, 9]) transforms a set of deduction rules into highly regular compiled forms, which facilitates quantitative analysis of queries and efficient query evaluation; while the latter (chain-based query evaluation) explores set-oriented evaluation of each compiled chain with appropriate constraint transformation and pushing, reducing unnecessary or redundant computation, and judgement of termination. The method can be viewed as a natural extension to relational query evaluation method and an integration of a top-down evaluation (by starting with the query as a goal) and a bottom-up evaluation (by set-at-a-time evaluation without infinite looping and repeated computation of subgoals).

The paper is organized as follows. In Section 2, we analyze why it is interesting to explore chain-based query evaluation. In Section 3, we introduce the LogicBase project and outline the algorithms. In Section 4, we present some experimental results, summarize our discussion and outline the future development plan.

2 Why chain-based query evaluation?

Although recursions can be in complex forms, most recursions in practical applications can be compiled into *chain* or *chain*-like forms to which efficient query analysis and evaluation techniques can be explored [13, 12]. We examine the strength and limitation of the chain-based compilation and evaluation technique.

2.1 Capture of more bindings in query binding propagation

First, by compiling complex recursions into highly regular chain forms, the selection-pushing technique can capture more bindings in complex recursions than those using traditional rule rewriting techniques, such as the magic rule rewriting [1, 27, 2]. This is illustrated by the following example [13].

Example 2.1 Traditional rule rewriting techniques may encounter some difficulties in the propagation of bindings in some recursive rules [13], which is demonstrated in the analysis of the following recursion.

Suppose that a query "$? - r(c, c_1, Y)$" is posed on a linear recursion defined by $\{(1), (2)\}$, where c's are constants, X's and Y's are variables, and r is a recursive predicate defined by EDB predicates a, b and e.

$$r(X_1, X, Y) \leftarrow e(X_1, X, Y). \tag{1}$$
$$r(X_1, X, Y) \leftarrow a(X, Y), r(X_2, X_1, Y_1), b(X_2, Y_1). \tag{2}$$

Following the binding propagation rules [27, 2], the bindings in the adorned goal, r^{bbf}, are propagated to the subgoal r in the body of the recursive rule, resulting in an adorned subgoal, r^{fbf}, as shown in (3), which are in turn propagated to the next expansion, resulting in r^{fff}, as shown in (4), which cannot propagate any bindings further to subsequent expansions, and the binding propagation terminates.

$$r^{bbf}(X_1, X, Y) \leftarrow a^{bf}(X, Y), r^{fbf}(X_2, X_1, Y_1), b^{bb}(X_2, Y_1). \tag{3}$$
$$r^{fbf}(X_2, X_1, Y_1) \leftarrow a^{bf}(X_1, Y_1), r^{fff}(X_3, X_2, Y_2), b^{bb}(X_3, Y_2). \tag{4}$$

This kind of binding propagation relies on the *backward* binding propagation only, in the sense that the bindings are propagated from the head to the body in a rule and from the IDB subgoal in the body of a rule to the head of its *unifying rule* (the rule which unifies it). For this recursion, the propagation cannot reduce the set of data to be examined in the semi-naive evaluation because the derived magic set contains the entire data relations. Furthermore, it is easy to verify that reordering of the subgoals cannot improve the evaluation efficiency.

For such recursions, binding information should be propagated in both forward and backward directions. That is, bindings should also be propagated *forward* from the body to the head in a rule and from the rule unifying the IDB subgoal to the corresponding IDB subgoal in the body of the original rule. Such a propagation cannot be caught by the traditional approaches but can be captured by the compilation (or normalization) of linear recursions [13].

For this recursion, its normalized, equivalent form is presented below.

$$r(X_1, X, Y) \leftarrow e(X_1, X, Y). \tag{5}$$
$$r(X_1, X, Y) \leftarrow a(X, Y), t(X_1). \tag{6}$$
$$t(X_1) \leftarrow a(X_1, U), b(X_2, U), t(X_2). \tag{7}$$
$$t(X_1) \leftarrow e(X_2, X_1, Y_1), b(X_2, Y_1). \tag{8}$$

Obviously, the bindings of the query r^{bbf} can be propagated to any expansions in the normalized recursion. □

The detailed compilation technique is presented in [13], which shows that a *single linear recursion* (with one linear recursive rule and one or more nonrecursive rules) can be compiled, independent of query forms, into either a *bounded recursion* (a set of nonrecursive rules) or a *chain recursion* (a formula consisting of a single chain or a set of synchronous chains). Moreover, many application-oriented recursions can either be compiled into (i) *asynchronous chain recursions* [12] and be evaluated by partial transitive closure algorithms [15, 14], or (ii) be compiled into *synchronous chain* forms (e.g., the *same generation* recursion) [2] and be evaluated by *counting*, *magic sets* [1, 2], or other chain-based evaluation methods [9]. Furthermore, algebraic simplification can be performed on the compiled expressions, and quantitative analysis can be performed by incorporation of query constants, integrity constraints and database statistics.

2.2 Chain-following and chain-split evaluation

The second strength of the method is that systematic and quantitative analysis can be performed on the compiled recursions to generate efficient query evaluation plans, such as chain-following vs. chain-split evaluation.

Since many recursions can be compiled into chain forms, *chain-based evaluation* should be explored on the compiled recursions. Chain-based evaluation can be viewed as an extension to relational database query analysis and optimization techniques because a compiled chain consists of an infinite set of highly regular relational expressions. The compilation makes explicit the regularity of the operation sequences in a recursion, on which quantitative analysis and optimization can be explored systematically. Such a quantitative analysis, similar to the access path selection and query plan generation for relational queries, can be performed based on the characteristics of the compiled chains, query instantiations, inquiries, integrity constraints, and database statistics of extensional relations [9]. Notice that quantitative analysis has been incorporated in many other recursion handling methods to generate different query evaluation plans as well.

Although function-free recursions cover an interesting class of recursions in deductive databases, many recursions in practical applications contain function symbols, such as structured data objects, arithmetic functions, and recursive data structures (lists, trees, sets, etc.). By transforming functions into functional predicates, the compilation and evaluation techniques developed for function-free recursions can be extended to functional ones [9]. Furthermore, the method can be generalized to logical programs containing modularly stratified negation [20] and those with higher-order syntax and first-order semantics [5]. Therefore, compilation of recursions into chain and pseudo-chain forms represents a powerful program transformation technique which transforms recursion into simple, easily-analyzable forms and facilitates the application of efficient evaluation methods.

In general, the chain-based query evaluation method consists of *chain-following, chain-split, existence checking*, and *constraint-based evaluation* techniques.

The simplest chain-based evaluation is *chain-following evaluation*, which starts with a highly selective end of a chain (called the **start end**) and proceeds towards the other end of the chain (called the **finish end**) and then possibly to other chains. It simulates partial transitive closure processing in the case of single chain recursion [15, 14] and the counting method [1, 7] in the case of multiple chain recursion.

Example 2.2 The recursion *length* defined by {(9), (10)} can be compiled into a double-chain recursion. For the query "? $-$ *length*$([a, b, c], N)$", the adorned normalized rule set is {(11), (12)}.

$$length([], 0). \tag{9}$$

$$length([X|L_1], succ(N_1)) \quad \leftarrow \quad length(L_1, N_1). \tag{10}$$

$$length^{bf}(L, N) \quad \leftarrow \quad L =^{bb} [], N =^{fb} 0. \tag{11}$$

$$length^{bf}(L, N) \quad \leftarrow \quad cons^{ffb}(X, L_1, L), length^{bf}(L_1, N_1), succ^{bf}(N_1, N). \tag{12}$$

The query can be evaluated by *counting* [1]. Starting at $L = [a, b, c]$, the *cons*-predicate is evaluated, that derives $L_1 = [b, c]$, and *count* (a variable in the *counting* implementation) is incremented by 1. The evaluation of the *cons*-chain terminates when $L_1 = []$ and *count* = 3. Then $N_1 = 0$ (obtained by the evaluation of the exit rule) initiates the *succ*-chain, which is evaluated *count* times and derives the length of the chain, $N = 3$. □

Depending on the available query bindings, *some* functional predicates in a chain generating path may not be immediately finitely evaluable, or the evaluation of a chain-generation path may generate a huge intermediate relation. In this case, a chain generating path can be partitioned into two portions: *immediately evaluable portion* and *buffered portion.* The former is evaluated but the latter is buffered until the *exit portion* (the expression which corresponds to the body of the exit rule) is evaluated. Then the evaluation proceeds in a way similar to the evaluation of a multi-chain recursion, except that the corresponding buffered values should be patched in the latter evaluation. Such an evaluation technique is called **chain-split evaluation** [9]. Here is one such example.

Example 2.3 The recursion *append* defined by {(13), (14)} can be compiled into a single-chain recursion. For the query "?$-$*append*$(U, V, [a, b])$" whose adorned predicate is $append^{ffb}$, the adorned normalized rule set is {(15), (16)}.

$$append([], L, L). \tag{13}$$
$$append([X|L_1], L_2, X|L_3]) \quad \leftarrow \quad append(L_1, L_2, L_3). \tag{14}$$

$$append^{ffb}(U, V, W) \quad \leftarrow \quad U =^{fb} [], V =^{fb} W. \tag{15}$$
$$\begin{aligned} append^{ffb}(U, V, W) \quad \leftarrow \quad & cons^{ffb}(X_1, W_1, W), append^{ffb}(U_1, V, W_1), \\ & cons^{bbf}(X_1, U_1, U). \end{aligned} \tag{16}$$

Since the chain "$cons(X_1, U_1, U)$, $cons(X_1, W_1, W)$" cannot be finitely evaluated as a whole based on the only available binding on W, a chain-split evaluation technique should be applied in the evaluation. That is, the chain should be split into two portions: (1) the *immediately evaluable predicate* "$cons(X_1, W_1, W)$", and (2) the *buffered predicate* "$cons(X_1, U_1, U)$".

The evaluation proceeds as follows. The evaluation of the exit rule derives the first set of answers: "$U = []$" and "$V = [a, b]$". The evaluation of the recursive rule proceeds along the immediately evaluable predicate "$cons(X_1, W_1, W)$" which derives "$W_1 = [b]$" and "$X_1 = a$" from "$W = [a, b]$". Then X_1 is buffered, and W_1 is passed to the exit rule, making "$V = [b]$" and "$U_1 = []$". Then the buffered U-predicate becomes evaluable since X_1 and U_1 are available. The evaluation derives "$U = [a]$". Thus, the second set of answer is $\{U = [a]$, $V = [b]\}$. Similarly, the evaluation may proceed along the immediately evaluable predicate "$cons(X_1, W_1, W)$" further, which derives the third set of the answer: $\{U = [a, b], V = []\}$. □

2.3 Constraint pushing and existence checking evaluation

Beside the distinction of chain-following vs. chain-split evaluation, another important strength of the method is the systematic analysis and exploration of available constraints.

Taking the evaluation of a single-chain recursion as an example, we examine how to push query constraints (or instantiations) at *both* ends of a compiled chain. The processing should start at a more restrictive end (the *start end*) and proceeds to a less restrictive end (the *finish end*). It is straightforward to push query constraints at the start end of the chain. However, care should be taken when pushing query constraints at the finish end.

Example 2.4 An IDB predicate $travel(FnoList, Dep, Arr, Fare)$, defined by {(17), (18)}, represents a sequence of connected flights with the initial departure city Dep, the final arrival city Arr, and the total fare $Fare$, where edb_flight is an EDB predicate representing the stored flight information.

$$travel([Fno], Dep, Arr, Fare) \leftarrow edb\_flight(Fno, Dep, Arr, Fare). \tag{17}$$
$$\begin{aligned} & travel([Fno|FnoList], Dep, Arr, Fare) \leftarrow \\ & \quad edb\_flight(Fno, Dep, Int, F_1), travel(FnoList, Int, Arr, F_2), Fare = F_1 + F_2. \end{aligned} \tag{18}$$

The recursion can be compiled into a single-chain recursion $\{(19), (20)\}$.

$$travel(L, D, A, F) \leftarrow$$
$$edb\_flight(Fno, D, A, F), cons(Fno, [], L), sum(F, 0, F). \tag{19}$$
$$travel(L, D, A, F) \leftarrow$$
$$edb\_flight(Fno, D, I, F_1), sum(F_1, S_1, F), cons(Fno, L_1, L), travel(L_1, I, A, S_1). \tag{20}$$

Suppose a query is to *find a set of (connecting) edb_flights from Vancouver to Zurich (Switzerland), with at most 4 hops and with the total fare between \$500 to \$800*, that is,

$$? - travel(FnoList, vancouver, zurich, F),$$
$$F \geq 500, F \leq 800, length(FnoList, N), N \leq 4.$$

According to the compiled form, D, L and F are located at one end of the chain (called the *departure end*); whereas A, L_1 and S_1 are at the other end of the chain (called the *arrival end*). The information at the departure end is, (i) $D =$ "*vancouver*", (ii) $500 \leq F \leq 800$, and (iii) $FnoList = L, length(FnoList, N), N \leq 4$; whereas that at the arrival end is, (i) $A =$ "*zurich*", (ii) $L_1 = []$, and (iii) $S_1 = 0$.

Since the information at the arrival end is more selective than that at the departure end, the arrival end is taken as the *start end.* Thus, all the query constraints at this end are pushed into the chain for efficient processing.

The query constraints associated with the finish end cannot be pushed into the chain in iterative evaluation without additional information. For example, pushing the constraint, "$Fare \geq 500$", into the chain will cut off a promising connection whose first hop costs less than 500. On the other hand, it is clearly beneficial to push the constraint, "$Fare \leq 800$", into the chain to cut off the hopeless connections when the accumulative fare is already beyond 800. However, a constraint like "$Fare = 800$" cannot be pushed into the chain directly, but a transformed constraint, "$Fare \leq 800$", can be pushed in for iterative evaluation.

A systematic way to push query constraints at the finish end can be derived by examining the interactions between query constraints and monotonicity constraints [9]. If the value (or the mapped value) of an argument in the recursive predicate monotonically increases but does not converge to a limit during the evaluation, a query constraint which blocks such an increase is useful at reducing the search space in iterative evaluation.

Based on the monotonicity constraint of the argument $Fare$, a *termination restraint template*, "$Fare \not> C$", is set up, where C is a variable which can be instantiated by a *consistent* query constraint. For example, a constraint, "$Fare \leq 800$", or "$Fare = 800$", instantiates the template to a *concrete termination restraint*, "$Fare \not> 800$". However, the constraint, "$Fare \geq 500$", is not consistent with the termination restraint template. Thus, it cannot instantiate a termination restraint. An instantiated termination restraint can be pushed into the chain for efficient processing.

Similarly, a constraint, "$Dep =$ '*vancouver*'", can be used for constraint pushing if we have the airport location information and a constraint: *same flight direction* (a monotonic constraint on flight direction). A concrete termination restraint, such as "$longitude(Dep) \not>$

longitude(*vancouver*)", can be derived from the analysis of the query constraints and monotonicity constraints of the recursion, and the tuples generated at any iteration with the departure airports located to the west of *Vancouver* is pruned in the chain processing. Also, the constraint, "*length*(*FnoList*, N), $N \leq 4$", can be pushed into the chain in the iterative evaluation. □

Notice that because of the availability of compiled chains and their precise connection information, it is straightforward to perform a detailed analysis of the monotonicity behavior of each chain and perform appropriate constraint transformation and constraint pushing for efficient evaluation. It is difficult to do so without precise compiled chain information.

Furthermore, with the availability of compiled chain information, it is easy to apply **existence checking evaluation** [9], which terminates the evaluation without an exhaustive search at the finish end of the chain if the query requires only to validate the existence of some answer to this end of the chain. This is illustrated by the following example.

Example 2.5 The recursion "$member(X, L)$", defined by {(21), (22)}, is a single-chain recursion by normalization. The normalized rule set with the adorned head "$member^{bb}(X, L)$" is {(23), (24)}.

$$member(X, [X|L_1]). \tag{21}$$

$$member(X, [Y|L_1]) \leftarrow member(X, L_1). \tag{22}$$

$$member^{bb}(X, L) \leftarrow cons^{bfb}(X, L_1, L). \tag{23}$$

$$member^{bb}(X, L) \leftarrow cons^{ffb}(Y, L_1, L), member^{bb}(X, L_1). \tag{24}$$

A query, "$?-member(a, [b, a, c, d])$", can be evaluated by an existence checking evaluation algorithm because the variables at both ends of the chain are instantiated but not inquired. Notice that "$L = [b, a, c, d]$" must be the start end, otherwise the *cons*-predicate is not finitely evaluable.

The evaluation proceeds as follows. The evaluation of the exit rule derives no answer since "$L_1 = [a, c, d]$", but "$b \neq X$". The evaluation of the recursive rule derives "$L_1 = [a, c, d]$", and "$member(a, [a, c, d])$". It is evaluated to *true* since "$member(a, [a, c, d])$" satisfies the exit rule. The evaluation terminates because one *true* answer validates the query. The evaluation of the recursive rule has to proceed until "$L_1 = []$" in (24) *only if* L contains no element a. In this case, the answer to the query is *false*. □

2.4 Limitations of chain-based query evaluation method

Since a (single) linear recursion can be compiled into a chain form or a bounded recursion [13], chain-based evaluation can be applied to such a recursion. Similarly, a **nested linear recursion** (which is a linear recursion in which some subgoal in the rule is defined in turn by a linear recursion) can also be compiled so and be evaluated by chain-based evaluation. For example, the popular n-queens recursion "$nqueens(N, Qs)$" [25] is a typical nested linear recursion, and efficient evaluation plans can be generated by our method for different query bindings, such as "$? - \ nqueens(8, Qs)$." and "$? - \ nqueens(N, [3,1,4,2])$.", independently of the ordering of rules and subgoals in the recursion definition.

Many complex recursions, though cannot be compiled into highly regular chains, may still have interesting regularities among the variable connections in the recursive rules. For example, the *tower_of_hanoi* recursion "$hanoi(N, A, B, C, Moves)$" [25] is a typical nonlinear recursion which cannot be compiled into highly regular chain forms. However, because of the regularity of its binding passing across two recursive subgoals in the recursive rule, the expansions of the recursive rule still demonstrate certain chain-like regularity and the portion in front of or behind each recursive subgoal in subsequent expansions can be treated as a pseudo-chain in the query analysis. Thus, the chain-based query evaluation method can still be applied to such recursions, and queries such as "$? - \ hanoi(3, a, b, c, Moves)$" or "$? - \ hanoi(N, a, b, c, [a\ to\ b, a\ to\ c, b\ to\ c, a\ to\ b, c\ to\ a, c\ to\ b, a\ to\ b])$." can still be analyzed systematically and be evaluated efficiently [11].

However, this does not imply that chain-based evaluation can be applied effectively to all kinds of recursions. This is because some recursions may not have regular variable passing patterns and cannot be compiled into chain or even pseudo-chain forms. For example, the nonlinear recursion r, defined by $\{(25), (26)\}$, belongs to this class.

$$r(X, X_1, Y) \leftarrow a(X, Y), r(X_1, X_2, Y_1), r(X_2, X_3, Y_2), b(X_3, Y_1, Y_2). \qquad (25)$$
$$r(X, X_1, Y) \leftarrow e(X, X_1, Y). \qquad (26)$$

Thus a major limitation of the chain-based evaluation method is its limited applicability to complex classes of irregular recursions.

In principle, chain-based evaluation is applicable to only a small subset of all the possible recursions. However, in practice, it is difficult to find a meaningful irregular recursion to which chain-based evaluation is inapplicable. This puzzling phenomenon could possibly be explained by the simplicity and regularity of human's thinking in writing recursive programs: A recursive program with no obvious expansion regularities is difficult for human to comprehend. Based on this phenomenon and the fact that chain-based evaluation may take advantages of the regularities of recursion structures to generate efficient query evaluation plans, our design of LogicBase takes chain-based evaluation as a major evaluation technique and leaves a more general technique, such as the generalized magic sets method, as an assistant one and be applied only when chain-based evaluation cannot derive efficient query evaluation plans.

3 Design and implementation of LogicBase

As a testbed for deductive query evaluation, a deductive database system prototype LogicBase has been designed and is being implemented in Simon Fraser University. The project is focused on the efficient evaluation of both function-free and function-bearing (functional) recursions. In comparison with the advanced features and extensions beyond recursive query evaluation, such as modularly stratified negation, negation with well-founded semantics, recursion with aggregation, etc. which have been implemented in many other deductive database systems [21], LogicBase can only be considered as a primitive prototype. Extensions to the system to include those features are in the future development plan.

3.1 Major algorithms: compilation and chain-based query evaluation

The current implementation of the LogicBase project consists of two major function blocks: (1) deduction rule compilation, and (2) chain-based query analysis and evaluation.

3.1.1 Deduction rule compilation

Deduction rule compilation consists of two major units: (1) *classification* (classification and simplification of recursions), and (2) *compilation* (compilation and normalization of recursions).

The classification unit takes a complex recursive program as input, rectifies it, eliminates mutual recursions when possible, simplifies the recursion when appropriate, and identifies the class of recursions to which the program belongs [12, 10]. By this processing, a recursion is classified into one of the following classes: (1) (single) linear recursion, (2) nested linear recursion, (3) multiple linear recursion, (4) regular nonlinear recursion, and (5) irregular recursion [12, 10].

The compilation unit takes the preprocessed recursion and compiles (normalizes) it into a chain program, when possible, based on a compilation (normalization) algorithm described in [13]. The compiled recursion is stored in the system for later query analysis and query evaluation.

3.1.2 Chain-based query evaluation

LogicBase implements a few interesting algorithms besides chain-based evaluation method to accommodate different kinds of recursions. Based on the class of a compiled recursion, a query evaluation method is selected by the following algorithm.

Algorithm 3.1 *Selection of a query evaluation method for a compiled recursion.*

Input. A compiled recursion.

Output. A selected query evaluation method.

Method.

```
CASE recursion OF
    • Function-free
        – asynchronous chain recursion: $\delta$-wavefront algorithm.
```

  - **synchronous chain recursion**: the magic sets method.
  - **otherwise**: the generalized magic sets method.
- Function-bearing /* containing function symbols, also called **functional** recursion. */
  - **when chain-based evaluation applicable**: chain-based evaluation.
  - **otherwise**: the generalized magic sets method.

END □

Rationale (for Algorithm 3.1).

Since function-bearing recursions have many different characteristics from function-free ones, we separate the two kinds of recursions and apply different evaluation methods.

For a function-free recursion, if it is compiled into an asynchronous chain recursion [12], a simple but relatively efficient partial transitive closure algorithm, δ-wavefront algorithm [12], is applied. There have been many other efficient database-oriented partial transitive closure algorithms developed recently (such as [14, 15]), which could be applied here for further performance improvement. If it is compiled into a synchronous chain recursion (e.g., the same-generation recursion), the magic sets method applies [2]. Otherwise (such as a nonlinear recursion), the generalized magic sets method applies [4].

For a function-bearing recursion, we apply chain-based evaluation, when applicable, and a generalized magic sets method, otherwise. Since a functional recursion does not have a finite least fixed point in general, the evaluation using the generalized magic sets method terminates when reaching a specified maximum number of iterations. However, since we have not found a practically interesting functional recursion to which chain-based evaluation is inapplicable, this algorithm is left there only for handling some unexpected cases. □

Obviously, the core of the evaluation of a function-bearing recursion is the chain-based query evaluation algorithm which is presented as follows.

Algorithm 3.2 *Chain-based evaluation of a normalized (functional) chain recursion.*

Input. (1) A normalized (functional) chain recursion, (2) a set of integrity constraints, (3) a query predicate, and (4) a set of query constraints.

Output. A query evaluation plan which incorporates the query constraints.

Method.

1. Test whether the query is finitely evaluable and terminable (using the algorithm presented in [8]). If it is not, stop.

2. Perform query binding analysis and determine (1) the start point of the chain processing, and (2) the predicate evaluation order. The analysis is based on (1) the available query bindings, (2) integrity constraints, (3) the structure of the recursion, (4) predicate selectivity based on the provided query and integrity constraints, and (5) the

average fan-out ratio of a predicate. Immediate evaluability and evaluation efficiency are the two major concerns in the ordering of the predicates.

3. Apply the query constraints associated with the start end as query instantiations to reduce the size of the initial set. If the whole chain is not immediately evaluable, the chain should be split into two portions with the immediately evaluable portion evaluated and the remaining portion buffered until the finish of the evaluation of the exit portion (*chain-split* evaluation). If the finish end of the chain is not inquired in the query, apply the *existence checking* evaluation.

4. If the finish end has associated query constraints, and there are monotonicity constraints available, apply the *constraint-based* evaluation by pushing the constraint or the transformed constraints, when possible, into the chain predicate(s) for efficient evaluation. Notice that these evaluation techniques can be combined: e.g., a constraint-based, chain-split evaluation can be applied if the query satisfies the conditions of both evaluation algorithms.

5. The query constraints which has not been used during the iterative evaluation of the recursion should be used at the end of the iterative evaluation. □

Rationale (for Algorithm 3.2).

Step 1 is necessary since a query must be finitely evaluable and terminable. It is performed using the methods discussed in [8]. Step 2 is necessary and correct since the correct ordering of the predicates will lead to finite and efficient evaluation. Step 3 is correct because the iterative query evaluation can be determined by a query binding analysis. The most selective information should be pushed into the compiled chain for initial processing [3]. If a chain is not immediately evaluable, the chain-split evaluation should be applied based on the query binding analysis [9]. If the finish end of the chain is not inquired in the query, the existence checking evaluation can be applied since it is adequate to find one answer which validates the chain. Step 4 is correct since the constraint information associated with the finish end should be maximally used when possible to reduce the search space. Step 5 is obviously necessary since the remaining query constraints, if not applied before, must be applied at the end of iterative processing to satisfy the query. □

The algorithm explores different evaluation directions, predicate ordering, the regularity and the structure of a recursion, and the maximal usage of query constraints and integrity constraints. The step-by-step analysis of sophisticated queries on a normalized chain recursion and the generation of efficient query evaluation plans provide high promise on the efficient evaluation of such programs. However, the optimality of the algorithm is not claimed here because the actual generation and selection of the query evaluation plan is a sophisticated and costly process, exponential to the size of the set of predicates in the recursion and the number of available access paths [16]. Similar to the dynamic query plan generation and query optimization in relational systems [27], one can only expect to derive suboptimal query evaluation plans at a reasonable cost of query optimization [2, 16].

4 Summary

The LogicBase system is being implemented based on the compilation and query evaluation algorithms described in the previous section. The system identifies different classes of recursions, eliminates mutual recursions when possible, and compiles recursions into chain recursions when appropriate. Queries posed to the compiled recursions are analyzed systematically with efficient query evaluation plan generated. Queries are executed mainly by chain-based evaluation, together with several other query evaluation methods, such as the generalized magic-sets method [4], the δ-wavefront algorithm [12], etc. The system is being tested on different kinds of recursions and queries with interesting experimental results and good performance. The system will be demonstrated in the workshop, using some interesting test programs, including many logic programs from Prolog textbooks [25].

As indicated in the introduction section, the current implementation of the LogicBase system is a primitive testbed for evaluation of deductive queries. Except for deductive query evaluation, most other features in many deductive database systems have not been incorporated in the implementation of LogicBase. The following two major features are planned to be incorporated in the future development of the system.

- Aggregation and modularly stratified negation.
- Towards a deductive and object-oriented database system.

There have been many interesting studies on the incorporation of these features in the framework of the magic-sets and query-subquery methods [19, 22, 17, 21, 24]. The incorporation of these features in the framework of chain-based evaluation is under investigation.

In summary, the LogicBase prototype system may represent an interesting alternative to efficient query evaluation in deductive database systems and may be worth further examination and development in deductive database research. This is our motivation to propose a demonstration of the LogicBase system prototype in the "Logic Database" workshop.

References

[1] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proc. 5th ACM Symp. Principles of Database Systems*, pages 1–15, Cambridge, MA, March 1986.

[2] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. 1986 ACM-SIGMOD Int. Conf. Management of Data*, pages 16–52, Washington, DC, May 1986.

[3] C. Beeri, P. Kanellakis, F. Bancilhon, and R. Ramakrishnan. Bounds on the propagation of selection into logic programs. In *Proc. 6th ACM Symp. Principles of Database Systems*, pages 214–226, San Diego, CA, March 1987.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proc. 6th ACM Symp. Principles of Database Systems*, pages 269–283, San Diego, CA, March 1987.

[5] W. Chen, M. Kifer, and D. S. Warren. Hilog as a platform for a database language (or why predicate calculus is not enough). In *Proc. Second Int. Workshop on Database Programming Languages*, pages 315–329, Gleneden Beach, OR, June 1989.

[6] D. Chimenti, R. Gamboa, R. Krishnamurthy, S. Naqvi, S. Tsur, and C. Zaniolo. The LDL system prototype. *IEEE Trans. Knowledge and Data Engineering*, 2:76–90, 1990.

[7] J. Han. Multi-way counting method. *Information Systems*, 14:219–229, 1989.

[8] J. Han. Constraint-based reasoning in deductive databases. In *Proc. 7th Int. Conf. Data Engineering*, pages 257–265, Kobe, Japan, April 1991.

[9] J. Han. Compilation-based list processing in deductive databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Extending Database Technology - EDBT'92 [Lecture Notes in Computer Science 580]*, pages 104–119. Springer-Verlag, 1992.

[10] J. Han. Compilation and evaluation of linear mutual recursions. *Information Sciences*, 69:157–183, 1993.

[11] J. Han and L. V. S. Lakshmanan. Evaluation of regular nonlinear recursions by deductive database techniques. In *SFU CSS/LCCR Technical Report TR93-09*, Simon Fraser University, July 1993.

[12] J. Han and W. Lu. Asynchronous chain recursions. *IEEE Trans. Knowledge and Data Engineering*, 1:185–195, 1989.

[13] J. Han and K. Zeng. Automatic generation of compiled forms for linear recursions. *Information Systems*, 17:299–322, 1992.

[14] Y. E. Ioannidis and R. Ramakrishnan. Efficient transitive closure algorithms. In *Proc. 14th Int. Conf. Very Large Data Bases*, pages 382–394, Long Beach, CA, August 1988.

[15] B. Jiang. A suitable algorithm for computing partial transitive closures. In *Proc. 6th Int. Conf. Data Engineering*, pages 264–271, Los Angeles, CA, February 1990.

[16] R. Krishnamurthy and C. Zaniolo. Optimization in a logic based language for knowledge and data intensive applications. In *Extending Database Technology (EDBT'88) [Lecture Notes in Computer Science 303]*, pages 16–33, Springer-Verlag, 1988.

[17] A. Lefebvre. Towards an efficient evaluation of recursive aggregates in deductive databases. In *Proc. Int. Conf. Fifth Generation Computer Systems*, pages 915–925, Tokyo, Japan, June 1992.

[18] S. Morishita, M. Derr, and G. Phipps. Design and implementation of the Glue-Nail database system. In *Proc. 1993 ACM-SIGMOD Conf. Management of Data*, pages 147–156, Washington, DC, May 1993.

[19] I.S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *Proc. 16th Int. Conf. Very Large Data Bases*, pages 264–277, Brisbane, Australia, August 1990.

[20] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. Coral - control, relations and logic. In *Proc. 18th Int. Conf. Very Large Data Bases*, pages 547–559., Vancouver, Canada, August 1992.

[21] R. Ramakrishnan and J. D. Ullman. A survey of research on deductive database systems. In *Journal of Logic Programming*, (to appear), 1993.

[22] K. Ross and Y. Sagiv. Monotonic aggregation in deductive databases. In *Proc. 11th ACM Symp. Principles of Database Systems*, pages 114–126, San Diego, CA, June 1992.

[23] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: Achievements and opportunities. *Comm. ACM*, 34:94–109, 1991.

[24] D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proc. 19th Int. Conf. Very Large Data Bases*, Dublin, Ireland, August 1993.

[25] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.

[26] S. Tsur. Deductive databases in action. In *Proc. 10th ACM Symp. Principles of Database Systems*, pages 142–153, Denver, CO, May 1991.

[27] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Vols. 1 & 2*. Computer Science Press, 1989.

[28] J. D. Ullman. A comparison of deductive and object-oriented database systems. In C. Delobel et. al., editor, *Deductive and Object-Oriented Databases (DOOD'91) [Lecture Notes in Computer Science 566]*, pages 263–277. Springer Verlag, 1991.

[29] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, and P. Stuckey. An introduction to the ADITI deductive database system. *Australian Computer Journal*, 23:37–52, 1991.

[30] L. Vieille, P. Bayer, V. Kuchenhoff, and A. Lefebvre. EKS-V1, a short overview. In *AAAI-90 Workshop on Knowledge Base Management Systems*, Boston, MA, July 1990.

The Aditi Deductive Database System

Jayen Vaghani, Kotagiri Ramamohanarao, David Kemp, Zoltan Somogyi, Peter Stuckey, Tim Leask, James Harland
Department of Computer Science, University of Melbourne
Parkville, 3052 Victoria, Australia
{jayen,rao,kemp,zs,pjs,tsl,jah}@cs.mu.oz.au

Aditi[1] is a deductive database system which has been developed at the University of Melbourne. [2] Programs in Aditi consist of base relations (facts) together with derived relations (rules), and are in fact a subset of (pure) Prolog. Queries are a conjunction of atoms (as in Prolog), and a bottom-up evaluation technique is used to answer queries. In finding all answers to a given query, Aditi, like many deductive database systems, uses algorithms and techniques developed for the efficient answering of queries in relational database systems. Thus we expect that Aditi need not be less efficient than a relational system for purely relational queries. Aditi also uses several optimisation techniques which are peculiar to deductive databases, particularly for the evaluation of recursive rules. These techniques include magic sets, supplementary magic sets, semi-naive evaluation, predicate semi-naive evaluation, the magic sets interpreter and the context transformation.

Aditi is based on a client/server architecture, in which the user interacts with a front-end process, which then communicates with a back-end server process which performs the database operations. There are three kinds of server process in Aditi: the query server, which manages the load that Aditi places on the host machine, database access processes, one per client, which control the evaluation of the client's queries, and relational algebra processes, which carry out relational algebra operations such as joins, selections and projections on behalf of the database access processes.

There are four main characteristics of Aditi which, collectively, distinguish it from other deductive databases: it is disk-based, which allows relations to exceed the size of main memory; it supports concurrent access by multiple users; it exploits parallelism at several levels; and it allows the storage of terms containing function symbols. It has been possible for researchers to obtain a beta-test version of Aditi since January 1993, and a full release of the system is expected soon. The current version of Aditi comes with a Prolog-like (text-based) interface, a graphical user interface, interfaces to both SQL and Ingres and a programming interface to Nu-Prolog. It is also possible to embed top-down computations within Aditi code.

A beta-test version of Aditi is available to interested researchers, and a full release of the system is expected soon.

---

[1]Aditi is named after the goddess in Indian mythology who is "the personification of the infinite" and "mother of the gods".

[2]This research is supported in part by the Australian Research Council, the Center for Intelligent Decision Systems, and the Collaborative Information Technology Research Institute.

The CORAL Deductive Database System*

Raghu Ramakrishnan   Praveen Seshadri†
Divesh Srivastava   S.Sudarshan‡

CORAL [2, 3] is a deductive database system that supports a powerful declarative query language. The language supports general Horn clause logic programs, extended with SQL-style grouping, set-generation, and negation. Programs can be organized into independently optimized modules, and users can provide optimization hints in the form of high-level annotations. The system supports a wide variety of optimization techniques. There is an interface to C++ that enables programs to be written in a combination of imperative and declarative styles. A notable feature of the CORAL system is that it is extensible. In particular, new data types can be defined, and new relation and index implementations can be added. An interface to the EXODUS storage manager [1] provides support for disk-resident data (both base and derived relations can be disk-resident), transactions and crash-recovery.

CORAL is available as source code (C++) from *ftp.cs.wisc.edu* at no charge. Versions compatible with g++ and AT&T C++ are available; the system has been successfully ported to Decstations, Sun Sparcstations, and HP workstations, to our knowledge. CORAL is distributed with extensive documentation, including a tutorial user manual and a large suite of example programs. It is currently installed at over 125 sites and is being used for instruction as well as in research projects. The demonstration is intended to illustrate some of the important features of the system, to exhibit a wide range of programs that benefit from these features, and to emphasize that substantial applications can be and have been developed using CORAL.

The features demonstrated include the following:

- general recursive rules
- non-stratified negation, aggregation and set-generation
- non-ground data structures such as difference-lists
- C++ interface, and its use for extensibility
- support for modules
- disk-resident base and derived data
- a graphical explanation package
- a stock market analysis system built as a CORAL application

We also illustrate the following issues related to program evaluation:

- program transformations such as Magic Templates (and variants) and Context Factoring.
- run-time techniques including Semi-Naive Evaluation (and variants) and Prolog-style "pipelined" evaluation.
- the use of high-level annotations to optionally guide optimization on a per-module basis.

With respect to performance, we show that:

- reasonable performance is achieved on a wide range of programs
- performance can often be tuned by a good organization of the program plus some annotations(hints)
- program compilation is fast, making interactive program development convenient

References

[1] M. Carey, D. DeWitt, G. Graefe, D. Haight, J. Richardson, D. Schuh, E. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. In *Readings in Object-Oriented Databases.* Morgan-Kaufman, 1990.

[2] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[3] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.

*This research was supported by a David and Lucile Packard Foundation Fellowship in Science and engineering, a Presidential Young Investigator Award, with matching grants from Digital Equipment Corporation, Tandem and Xerox, and NSF grant IRI-9011563.

†The address of the first two authors is Computer Sciences Department, University of Wisconsin, Madison, WI 53706, U.S.A. These authors' email addresses are {raghu, praveen}@cs.wisc.edu.

‡The address of the last two authors is AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A. These authors' email addresses are {divesh, sudarsha}@research.att.com.

Demonstrating the Glue-Nail Deductive Database System

Marcia A. Derr
AT&T Bell Laboratories
mad@research.att.com

Geoffrey Phipps
Sun Microsystems Laboratories
phipps@eng.sun.com

The Glue-Nail deductive database system [1, 2] provides two complementary languages for programming applications. The Nail declarative language is used to express simple and complex queries or views. The Glue procedural language augments queries with control structures, update operations, and input/output procedures. In this demonstration, we guide the audience through a typical session of compiling and running a Glue-Nail program.

A Glue-Nail program consists of one or more modules of Glue procedures and Nail rule sets. Each module can be compiled separately. The Glue compiler translates Glue procedures into the IGlue target language. It also extracts Nail queries and their associated rule sets and passes them to the Nail compiler. The Nail compiler translates its source into IGlue code, using appropriate transformation and evaluation strategies. The IGlue code that is generated by either compiler may be optionally analyzed and improved by the static code optimizer. In the final compilation step, the linker gathers all relevant IGlue code into a single file.

The IGlue interpreter loads the IGlue program and reads into memory any disk-resident relations that the program will access. As the interpreter executes each instruction, it calls the run-time optimizer, when appropriate, to adapt query plans to changing parameters of relations. When the interpreter halts, it writes to disk any persistent relations that have been updated. The interpreter includes tracing and profiling tools for demonstrating or debugging its operation.

The Glue-Nail system was developed at Stanford University as part of the NAIL! Project. The current implementation supports single-user applications with databases that fit in main memory. The system has been tested on a variety of applications including a logic simulator, a flight reservation system, scheduling and allocating resources for building construction, and bill-of-materials.

Availability: The front and back ends of the Glue-Nail system are distributed separately. Requests for the Glue and Nail compilers should be sent to Geoffrey Phipps. Requests for the IGlue interpreter should be sent to Marcia Derr.

References

[1] Marcia A. Derr, Shinichi Morishita, and Geoffrey Phipps. Design and implementation of the Glue-Nail database system. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993.

[2] Geoffrey Phipps, Marcia A. Derr, and Kenneth A. Ross. Glue-Nail: A deductive database system. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 308–317, Denver, Colorado, May 1991.

The XSB Programming System

K. Sagonas, T. Swift, and D.S. Warren
Dept. of Computer Science, Suny at Stony Brook

Version 1.4.0 of XSB, a Prolog-based Logic Programming System, is now available. XSB extends the standard functionality of Prolog to include implementations of SLG resolution (tabling) and of HiLog terms.

SLG resolution is useful for recursive query computation, allowing programs to terminate correctly in many cases where Prolog does not. Users interested in Parsing, Program Analysis, and memory-resident Deductive Database applications may benefit from XSB.

XSB's SLG implementation:

- Is incorporated at the emulator level for maximal efficiency. The speed improvement over meta-interpreters written by the XSB group is 2-3 orders of magnitude, with considerable improvements in space as well.
- Evaluates programs with (left-to-right modularly) stratified negation and aggregation at the engine level. The current version evaluates programs with general negation at the interpreter level.
- Allows for declaration of tabled predicates either automatically by the system or manually by the user.
- Provides standard tabling predicates which can be used to program a number of applications like meta-interpreters for the well-founded semantics (provided as an example program).
- Allows full Prolog functionality in tabled code, including cuts (subject to weak semantic restrictions), meta-logical predicates, 'alsecond-order predicates, etc.

HiLog supports a type of higher-order programming in which predicate symbols can be variable or structured. This allows unification to be performed on the predicate symbols themselves in addition to the arguments of the predicates.

XSB's HiLog implementation:

- Includes a fully integrated HiLog preprocessor. HiLog terms can be used anywhere in XSB, including the interpreter level.
- Includes compiled HiLog. Higher-order predicates execute at about a 50% overhead over comparable compiled first-order predicates.
- Provides a number of meta-logical standard predicates for HiLog terms.

The new version also offers novel indexing along with faster I/O than previous versions and emulator improvements. Users are offered the choice of Prolog-style hash-based indexing, or *first-string* indexing. In the hash-based indexing, users can index on various arguments or on multiple arguments, removing a limitation of Prolog for data-oriented queries. Furthermore, users can index clauses with first-string indexing which builds a discrimination net, and reduces the need for backtracking.

Version 1.4.0 of XSB has been tested on over a dozen hardware and operating system platforms. [1] It is available through anonymous ftp from `cs.sunysb.edu`, and through Gopher.

Queries can be directed to `xsb-contact@cs.sunysb.edu`.

[1] Currently SPARC, MIPS, Intel 80x86 and Motorola 680x0 chips have been tested; for operating systems, SUNOS, SOLARIS, IRIX, ULTRIX, LINUX, 386BSD, AMIGA-DOS, HP-UX, System V.3, SCO Unix, and Mach have been tested.

Outline of the LogicBase Demonstration[*]

Jiawei Han, Ling Liu and Zhaohui Xie
School of Computing Science
Simon Fraser University
Burnaby, B.C., Canada V5A 1S6
{han, lingliu, zhaohui}@cs.sfu.ca

The **LogicBase** demonstration consists of 2 sections:

1. **Compilation of complex linear recursions.**

   Complex linear recursion can be compiled into highly regular chain forms or normalized rewritten rules. The demo will show two examples:

   (a) $strange_1$, an example shown in the LogicBase paper. It requires the bi-directional binding propagation.

   (b) $strange_2$, another complex linear recursion consisting of mutliple V-units. It requires sophisticated efforts for normalization.

   There are more than 50 complex linear recursions stored in the same test subdirectory and workshop attendees may play with them after the presentation.

2. **Chain-based query evaluation for function-bearing linear recursions.**

   The following query evaluation examples will be shown in the demo. Notice that all the rules in a recursion and all the sub-goals in a rule are completely order-independent. One can change the orders in many ways and re-execute the programs. Due to the time limitation, we will show only a few executions with order-swapped programs.

   (a) **append**: Queries with different bindings on the *append* recursion, such as *append-bbf, append-bfb, append-ffb, append-bbb, append-fbf,* etc. will be tested on the same program with no *hints* to the system.

   (b) **nqueens**: Different queries on the same *nqueens* recursion (with no *hints* to the system), such as "$?{-}nqueens(5, Qs).$", "$?{-}nqueens(N, [4, 1, 3, 2]).$", "$?{-}nqueens(5, [5, 1, 3, 2, 4]).$" "$? - nqueens(N, Qs).$", etc. will be tested on the system.

   (c) **isort**: Different queries on the same *isort* recursion (*insertion sort*), such as "$? - isort([1, 5, 7, 2, 6], Qs).$", "$? - insort(L_1, [1, 2, 3, 4]).$", "$? - isort([1, 5, 2, 4], [1, 2, 5, 4]).$" "$? - nqueens(L_1, L_2).$", etc. will be tested on the system.

   (d) **flight**: The popular air-flight reservation example will show that different constants and constraints can be pushed smartly into the recursion for efficient evaluation.

There are many other interesting examples in the testing directories. The demo will show more examples when time permits. Workshop attendees are welcome to play with other examples after the demonstration.

[*]This work was supported in part by the *Natural Sciences and Engineering Research Council of Canada* under the grant OPG-3723 and a research grant from the *Centre for Systems Science of Simon Fraser University.*

Overview of the deductive database system *LOLA*

Burkhard Freitag, Heribert Schütz, Günther Specht

Institut für Informatik, Technische Universität München
Orleansstr. 34, D-81667 München, Germany
email: `{freitag,schuetz,specht}@informatik.tu-muenchen.de`

The centerpiece of the deductive database system prototype *LOLA* is a logic programming language with functions and negation, i.e. a proper superset of DATALOG. The *LOLA* language has an iterated fixpoint semantics and integrates deduction and efficient data access using relational techniques. *LOLA* provides automatic access to external relational database systems via a SQL interface. In addition, functions of the host language Common Lisp can be called from within *LOLA* programs and vice versa. There is no integrated update language. Updates to the base relations must be performed using the host language.

LOLA rules and queries are organized into *units* that consist of both definitions, i.e. a set of rules, and type declarations for every constant, function symbol, and predicate symbol occurring in this unit. Polymorphic types are supported.

As opposed to Prolog-like top-down interpreting systems, *LOLA* queries are compiled into a set-valued expression in a top-down phase. In a subsequent - possibly deferred - bottom-up phase this expression is evaluated and the appropriate set of answer tuples is returned. The basic evaluation scheme is semi-naive fixpoint iteration. Several optimizations, among others the magic set transformation, can optionally be applied. An explanation facility is integrated into the system.

The *LOLA* system is implemented in Common Lisp and has been tested on UNIX platforms with Allegro Common Lisp, AKCL, and Lucid Common Lisp. *LOLA* is available via FTP . Please direct requests to `lola-request@informatik.tu-muenchen.de`.

For more information on the *LOLA* system see the articles listed below.

References

[1] B. Freitag, H. Schütz, and G. Specht. *LOLA* — a logic language for deductive databases and its implementation. In *Proc. 2nd Intl. Symp. on Database Systems for Advanced Applications (DASFAA '91)*, pages 216 – 225, Tokyo, 1991.

[2] B. Freitag, H. Schütz, G. Specht, R. Bayer, and U. Güntzer. LOLA — ein deduktives Datenbanksystem. In R. Bayer, T. Härder, and P. Lockemann, editors, *Objektbanken für Experten*, Informatik Aktuell, pages 1 – 28. Springer-Verlag, Berlin, 1992. (in German).

[3] B. Freitag, H. Schütz, G. Specht, R. Bayer, and U. Güntzer. *LOLA* – a deductive database system with integrated SQL-database access. Technical report, Technische Universität München, Munich, Germany, 1993.

Extending Deductive Databases with Object Orientation

A Presentation of the ECRC/IDEA Year1-Demo

Stéphane Bressan, Willem Jonker, Andrea Sikeler
{steph,jonker,sikeler}@ecrc.de

The IDEA project at ECRC is focusing on the design and support of a high-level conceptual interface for a knowledge processing environment, based on an object-oriented conceptual model and a declarative rule language. The resulting system will provide advanced functionality to raise the quality of interaction with users and applications as well as the quality and efficiency of data management.

Part of the work is carried out in the context of the ESPRIT EP6333 project of the same name (started in June 1992 and projected for four years). The ESPRIT consortium comprises some of the most renown institutions and researchers in the database field: BULL (prime contractor), ICL, the Imperial Cancer Research Fund, INRIA, TXT Ingeneria Informatica, and the Politecnico di Milano.

Up till now the focus has been on the design of the high-level conceptual interface of the IDEA system. The developed interface extends the rule formalism by adding object orientation. ECRC is using its ECL^{i}PS^e logic programming environment as a basis for the environment to be developed. The persistent storage facilities offered by ECL^{i}PS^e supports efficient storage of deductive rules and will be augmented to serve as an internal low level object manager offering efficient storage facilities for complex objects. Also experiences from the development of the EKS-V1 system in the areas of query optimization and integrity constraints will guide the design decisions for the IDEA system.

It is in the context of the ESPRIT project that the IDEA technology will be validated by applications in three domains. ICRF (in collaboration with ECRC) will develop a decision support system for molecular biology researchers. The other application domains are software testing and electricity network control.

The current prototype is a fast first implementation and its sole purpose is to serve as a demonstrator for the kind of functionality that the final IDEA system will offer. Documentation and information can be obtained from:

Willem Jonker
European Computer-Industry Research Centre
Arabellastrasse 17
D-81925 Munich
Germany

Hy$^+$: A Hygraph-based Query and Visualization System

Mariano P. Consens Alberto O. Mendelzon

{consens,mendel}@db.toronto.edu
Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A1

The Hy$^+$ system [CM93] provides a user interface with extensive support for visualizing and querying structural (or relational) data as *hygraphs* [Con92], a convenient abstraction that generalizes several diagrammatic notations.

Hy$^+$ supports visualizations of the actual database instances and not just diagrammatic representations of the database schema. Given the large volume of data that the system must present to the user, it is fundamental to provide her with two fundamental capabilities: the ability to *define* new relationships (or derived data), and an innovative way of using queries to decide what data to *show*. Using Hy$^+$'s show capability the user can selectively restrict the amount of information to be displayed. This *filtering* of irrelevant data is fundamental if one is to have any hope of conveying manageable volumes of visual information to the user. Selective data visualization can be used to locate relevant information, to restrict visualization to interesting portions of the data, and to control the level of detail at which the information is presented.

To describe queries, Hy$^+$ relies on a visual pattern-based notation. The patterns are expressions of the GraphLog query language [Con89, CM90]. Overall, the system supports query visualization (i.e., presenting the description of the query using a visual notation), the (optional) visualization of the data that constitutes the input to the query, and the visual presentation of the result.

Hy$^+$ and GraphLog have been successfully applied in areas where it is helpful to visualize the data using hygraph based diagrams, such as: exploring C++ source code [CM93], formal software design documentation and object code overlay structure [CMR92]; browsing the structure of hypertext documents [CM89]; debugging distributed and parallel programs [CHM93]; and supporting network management [CH93].

The Hy$^+$ system is implemented as a front-end, written in Smalltalk, that communicates with other programs to carry on tasks such as data acquisition, query evaluation, hygraph layout [Noi93], and invoking external programs to browse the objects represented by the visualizations (i.e., editing source code in a software engineering application). The front-end provides browsers that let users interact with the hygraph-based visualizations, as well as supporting parsing, query translation, back-end communication and answer management. There are three back-end query processors used by the system: LDL [NT89], CORAL [Ram92], and a previously developed GraphLog interpreter implemented in Prolog [Fuk91]. The reader is referred to [CMV93] for a description of the use of deductive database technology in the Hy$^+$ system.

Acknowledgements

We would like to thank the other members of the Hy$^+$ development group for their contributions to the system: Frank Eigler, Yanni Jew, Masum Hasan, Emanuel Noik, Brian Sallans, Dimitra Vista, and Annie Yeung.

The Hy$^+$ project is supported by the Natural Sciences and Engineering Research Council of Canada, the Information Technology Research Centre, the Institute for Robotics and Intelligent Systems, and the IBM

Canada Centre for Advanced Studies. The first author was supported by an IBM Canada Research Fellowship and by PEDECIBA (Uruguay).

References

[CH93] Mariano Consens and Masum Hasan. Supporting network management through declaratively specified data visualizations. In *Proceedings of the Third IFIP/IEEE International Symposium on Integrated Network Management*, pages 725–738. IFIP Transactions C-12, Elsevier North-Holland, 1993.

[CHM93] Mariano Consens, Masum Hasan, and Alberto Mendelzon. Debugging distributed programs by visualizing and querying event traces. Abstract presented at the ACM/ONR Workshop on Parallel and Distributed Debugging, 1993.

[CM89] Mariano Consens and Alberto Mendelzon. Expressing structural hypertext queries in GraphLog. In *Proceedings of the Second ACM Hypertext Conference*, pages 269–292, 1989.

[CM90] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[CM93] Mariano Consens and Alberto Mendelzon. Hy$^+$: A hygraph-based query and visualization system. In *Proceedings of the ACM-SIGMOD 1993 Annual Conference on Management of Data*, pages 511–516, 1993. Video presentation summary.

[CMR92] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *14th. Intl. Conference on Software Engineering*, pages 138–156, 1992.

[CMV93] Mariano Consens, Alberto Mendelzon, and Dimitra Vista. Deductive database support for data visualization. Technical Report CSRI-285, University of Toronto, 1993. (In Declarative Database Visualization: recent papers from the Hy$^+$/GraphLog project, pages 51-66). Submitted to EDBT'94.

[Con89] Mariano P. Consens. Graphlog: "real life" recursive queries using graphs. Master's thesis, Department of Computer Science, University of Toronto, 1989.

[Con92] Mariano P. Consens. Visual manipulation of database visualizations. PhD Thesis Proposal, in preparation, 1992.

[Fuk91] Milan Fukar. Translating GraphLog into Prolog. Technical report, Center for Advanced Studies IBM Canada Limited, October 1991.

[Noi93] E.G. Noik. Graphite: A suite of hygraph visualization utilities. Technical Report CSRI-285, University of Toronto, 1993. (In Declarative Database Visualization: recent papers from the Hy+/GraphLog project, pages 108-126).

[NT89] Shamim Naqvi and Shalom Tsur. *A logical language for data and knowledge bases*. Computer Science Press, New York, 1989.

[Ram92] R. Ramakrishnan, D. Srivastava and S. Sudarshan. Coral: Control, relations and logic. In *Proc. Intl. Conference on Very Large Data Bases*, 1992.