— MegaLog —
A platform for developing Knowledge Base Management Systems

Jorge B. Bocca
ECRC
Arabellastr. 17
D-8000 München 81, Germany

Abstract. *This is an overview of* **MegaLog** *- a platform on which next generation Knowledge/Data Base Management Systems could be built. To achieve this purpose, the requirements of object oriented and of deductive K/DBMSs were considered in the design of MegaLog, and features to support them efficiently were built into it. This is indeed an assertion that there is no contradiction in the fundamental principles on which these two types of K/DBMS rest. On the contrary, there are many important elements in common, and those principles that are not common are at the very least complementary to each other.*

The original contributions in the design and implementation of MegaLog are threefold: the common platform approach discussed above, the techniques that make possible the persistence of programs and data for shared and concurrent usage on a large scale, and finally but not least important - the use, scale and scope to which conventional techniques in the fields of logic programming and of data bases have been applied.

1 Introduction

For a number of years now, research in the field of knowledge/data bases has focused on making the technology of modern programming languages available to knowledge/data base users. Research work on object oriented data bases has mainly concentrated on bringing high level concepts found in languages such as SmallTalk and C++ into the next generation of data base systems [34, 20, 27]. However this has not been an exclusive aim of the object oriented research community, but also of many other researchers. In a broader sense, one finds this trend in the research efforts of the knowledge/data base community at large. Newly developed systems, or systems still in the process of design clearly show this tendency. Systems derived from relational technology such as Starburst [18] and POSTGRES [29] are examples. The ultimate aim of this effort in our opinion is to totally eliminate the distinction in the treatment given to data and programs.

In the last few years, researchers in the field of persistent programming languages have sought to eliminate the language 'impedance mismatch' caused by the different treatment given to data and programs, by making the programming language persist [1]. Although significant progress has been made in this direction, the levels of performance of the experimental systems built so far, among other factors, rule them out for practical applications which require a scaling factor beyond the boundaries of current main memory technology.

At least in theoretical terms, logic programming provides an ideal basis to develop a persistent programming environment which in addition can support the high level concepts found in object oriented systems. At a simple level Zaniolo [35] has shown how this can be done. In particular, Horn clauses seen as a uniform representation for data and procedures allow for the manipulation of structurally highly complex terms (whether they are used as data or procedures is irrelevant). Moreover, logic programming provides one extra feature which is important by itself - **deduction.**

In general, deduction and object orientation are complementary to each other and their coming together in a common knowledge/data base platform is highly desirable. However, the issue of performance remains a fundamental one.

In **MegaLog** our objective has been to construct an efficient platform to provide support for navigational and set evaluation strategies simultaneously over highly complex data and/or procedures. In addition, we have sought to provide mechanisms with which to efficiently support the high level concepts found in object oriented and deductive systems. In our view, the fullfilment of these requirements not only makes possible the successful development of object oriented and/or deductive data base systems as the next technology in data base systems, but also allows the co-existence and sharing of data and programs amongst K/DBMSs based upon different data models. It also provides support for the emulation of applications developed for the older generation of relational and navigational data base systems.

The results obtained with the development of MegaLog rest on research work on a number of experimental systems that preceded it. In **Educe** [12, 10, 11, 9] we studied the problems of coupling and integrating logic programming systems with a relational DBMS. We showed the relevance of the technologies to the problem at hand. However, we also showed the inadequacies of coupling or integrations of existing systems - in particular, the inability of the DBMS to manage rules efficiently in persistent store. In Educe, we experimented with keeping rules in source form in the relational engine. In the **KB-Prolog** compiler [4, 5, 6] we designed and built the basic inference machinery to operate over data intensive applications for a Prolog system. In **Educe*** [13, 14] we expanded the functionality of KB-Prolog with a basic relational sub-system which was syntactically and semantically integrated with the inference engine. MegaLog and its programming language represent the latest stage of development of this work.

This paper introduces the subject with a presentation of the technological background of MegaLog. It continues with the general principles of design and an overview of the implementation. Then, it presents and discusses the techniques of incremental compilation of Horn clauses in the context of persistence and the mechanisms used to execute them. Conclusions are presented based on performance figures obtained from benchmarks and applications.

2 The background of MegaLog

Logic programming in general and the language Prolog in particular have proved their worth in practice in the implementation of deductive systems and expert systems of small scale. Because of these results, it has often been claimed that a good way in which to build a platform for

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

the next generation of KBMSs could be to use Prolog as a front-end to a relational DBMS [26, 12, 10, 9]. The mechanism suggested is either a coupling or an integration of a Prolog interpreter/compiler to an existing relational DBMS. It is our view that this is a rather poor solution, if any at all. The syntax and semantics of the language Prolog fall short of the requirements to scale up the technology. Three problem areas have already been recognized:

1. *Performance and reliability.* In the context of persistence, major deficiencies in performance are soon noticed by users of these hybrid systems. A particular aspect of this which has perhaps not until now been clearly isolated is the way in which the performance of these systems degrades to the point of collapse after a relatively short period of operation. This is caused by the inability of the component systems to share basic control information. For example, the appearance of a cut in a Prolog program should cause a release of buffers, cursors and other resources by the underlying DBMS.

2. *Power of expression and evaluation machinery.* The research work on so called *recursive* systems [2] points to yet worse situations. In a coupling/integration, the mismatch between the expressive power of the programming language and that of the evaluation machinery very often leads to programmers being allowed to express what the underlying execution machinery cannot properly evaluate.

3. *Mismatch of the run time support engines.* Similarly, the two independently developed mechanisms for run time support, i.e. the unification and relational engines, fail to solve practical problems such as the ones caused by the simultaneous use of two different data type systems and two different implementations of common types [12].

Because of these problems and encouraged by our experience developing Educe* [13, 14] we concluded that is was necessary to develop a new system from scratch, starting only from the principles of data base and logic programming technologies. However, to achieve the ultimate objective of providing the platform outlined in the introduction, some fundamental issues beyond the 'impedance mismatch' problem need to be addressed. Among the most important of them are: deduction rules; complex objects handling, encapsulation and inheritance; and evaluation strategies.

2.1 Rules in Persistent Store

Although relational DBMSs are particularly successful in handling simple factual information, their usability is at best questionable when the nature of the information to be handled increases in complexity. Thus for example, the capability to maintain rules in persistent and shareable store is beyond the functionality of relational DBMSs. To see the applicability of this facility, consider an airline data base. Say we want keep information on flights. For example, we want to state that there is a flight from London to Glasgow once every two hours, every working day of the week, between eight o'clock in the morning and eight o'clock in the evening. This can be expressed in MegaLog by use of a rule - the first clause in the deductive relation *connect*:

```
connect( london, glasgow, Day, Depart, Arrive) :-
        work_day(Day),
        period( 8:00, 20:00, 2:00, Depart),
        add_time( Depart, 0:50, Arrive).
connect( glasgow, dundee, wednesday, 11:20, 12:05).
    :
```

At this point, we depart from the vision of separating programs from facts (second clause) and we rely on deduction as a generalization of the concept of retrieval used in data bases. This, in our new formulation of the airline knowledge base, avoids the need to keep one entry in the *connect* relation for each flight between two cities. Now, only a few clauses in the *connect* relation are sufficient to summarize all of them [1].

More importantly though, if the airline decides to update the knowledge base, say by increasing the numbers of flights London - Glasgow from only work days (Monday to Friday) to daily, that is easily done by replacing the first clause above with:

```
connect( london, glasgow, Day, Depart, Arrive) :-
        day(Day),
        period( 8:00, 20:00, 2:00, Depart),
        add_time( Depart, 0:50, Arrive).
```

2.2 Objects

The concept of organizing/modelling knowledge around collections of objects and their mutual relationships is the central one in object-oriented languages. Communication between objects is allowed only by means of message passing. It is this paradigm that makes the implementation of individual objects independent of the implementation of other objects - in other words, a high level of modularity is achieved. Clearly, this characteristic is highly desirable in knowledge and data base management systems.

Similarly, because of its contribution to the organizing/modeling of knowledge, one would like to find support for class inheritance in knowledge and data base management systems. This makes possible the sharing of existing methods by newly created classes of objects.

MegaLog inherits from Prolog very rich facilities for dynamically building complex structures - logic terms. These terms give to users of MegaLog the possibility of representing complex objects. Similarly, efficient implementations of particular types of encapsulation and inheritance are supported at the level of the MegaLog abstract machine. Thus for example, at the architecture level, one abstract machine word maps to two words in the host machine running MegaLog. This extra support for tagging and classification, used in conjunction with other primitives provided by MegaLog, gives potential implementors of knowledge/data base systems a powerful mechanism for handling subtypes, inheritance and/or signature analysis. At a more simplistic level, it is also possible within MegaLog to define the operators **with** and **isa** proposed by Zaniolo [35] to implement inheritance and encapsulation. For example, in MegaLog one could define the class *air_route* and then, the sub-class *direct_air_route* wich inherits the methods of *air_route*[2]:

```
?- air_route( No) with
   [(new( From, To, Stops) :-
      setval_private( No, air_route,
                      a_route(From, To, Stops))),
    (route( Route) :-
      getval_private( No, air_route,
                      a_route(From, To, Stops)),
      conc( Stops, [To], X),
      conc( [From], X, Route) ),
    (delete :-
      erase_private_array(No, air_route/0))
   ],

   direct_air_route(No) with
   [(new(From, To) :-
      air_route(No):new( From, To, []))
   ].

?- direct_air_route(No) isa air_route( No).
```

Although the significance of the above concepts for the development and maintenance of knowledge/data bases is now widely accepted, the mechanisms provided in the current generation of object oriented data bases are in our view restrictive in two important aspects: only deterministic procedural methods are allowed; and intensional objects are not supported.

MegaLog not only makes possible the implementation of the functionality required to efficiently support the fundamental concepts of object

[2] setval_private/3, getval_private/3 and erase_private_array/2 are built-ins to deal with variables local to a procedure, but beyond the scope of an individual clause.

orientation. It eliminates the above restrictions through its facilities to dynamically build complex data structures - inherited from the language Prolog - and to store them persistently.

Also, because of MegaLog's roots in logic programming, it goes further beyond the current generation of object oriented data base systems, by providing features that in our view are fundamental. That is: to represent knowledge associatively; to incrementally update the schema of knowledge/data bases; and, to keep data and procedures together and treat them the same.

2.3 Evaluation Strategies

Lengthy discussions have been pursued in the literature about the inherent superiority in performance of set evaluation over tuple at a time evaluation [3, 12, 7, 9, 17, 26, 28, 31, 33]. In an early paper [12], we showed by empirical means that there is no clear cut answer to the above question. There are cases in which it is absolutely clear that set evaluation techniques should be used to obtain good performance, in preference to tuple at a time evaluation techniques. However, in the above study we also found significant cases, particularly in deductive and object oriented type of environments, where navigational searches proved the opposite to be true. Thus the next important issue is how to provide, at the user level, syntax that is regular, simple and uniform to provide both evaluation strategies within the one system. We achieve this in MegaLog by adopting a mechanism similar to the *is/2* predicate of Prolog for arithmetic to introduce the concepts of relation and deductive relation. Specific examples of programs using set and tuple at a time evaluation are given in the next section.

3 Persistent Logic Programming

MegaLog, as already mentioned in the previous section, has its roots in logic programming. It subsumes the language Prolog in a persistent manner. In addition to the power of expression given by Prolog as a general purpose host language, a number of new language constructs and facilities are offered. In this section, we describe the most significative of these facilities.

3.1 Base Relations

The purpose of providing the data type *base relation* (normally referred to as *relation*) is to give users of MegaLog the facilities and performance of a conventional relational system in a well integrated deductive context. A relation can be explicitly defined by use of the operator <=>. If for example, in our airline knowledge base, each individual flight were represented as a tuple in a relation, we would define such a relation by (notice the use of default values - Isz, Asz and Def):

```
flight <=> [ atom( from, 20, '+'),
          atom( to,   20, '+'),
          atom( day,  Asz, Def),
          integer( dep,  Isz, _),
          integer( arr,  Isz, _) ].
```

The relation could then be populated by means of the set operator for insertion <++ :

```
flight <++ [ ( munich, frankfurt, monday, 1000, 1200),
          ( frankfurt, london, sunday, 800, 900),
          ( london, glasgow, friday, 2000, 2045),
          ( glasgow, dundee, monday, 1230, 1255) ].
```

Suppose that after our relation *flight* has been populated, we wanted to list the departure and arrival times of all the flights from *London* to *Glasgow* departing on a *Monday* between *14:00* hrs and *17:00* hrs:

```
?- Flights isr [ dep, arr] :^: flight
     where dep =< 1400 and dep >= 1700 and from == london
     and to == glasgow and day == monday,
   printrel( Flights).
```

The variable *Flights* is instantiated to the name of the result relation (a temporary relation generated by the system). If the result were to be kept for further work later on, perhaps in a different session, then a permanent relation should be created by using an atom as a name, instead of the variable *Flights*. In the formulation of the query, the use of the projection operator :^: should also be noticed.

The relational algebra in MegaLog is regular and orthogonal, so any relational expression that is allowed on the right hand side of the set retrieval **isr** operator can likewise be used with the set operators for insertion <++ and deletion <--. Thus, for example, to enforce an integrity constraint we might want to delete any entry from our *early_london_bound* relation that arrives in *London* after *10:00* hrs:

```
?- early_london_bound <-- [from, day, dep, arr] :^: flight
     where arr > 1000
     and   to == london.
```

However powerful and efficient the algebra might be in evaluating universally quantified queries, there are still the existential queries for which set evaluation techniques are grossly inefficient. Suppose we wanted simply to know if there is an air connection between *Glasgow* and *Dundee*:

```
/* ignore date and time */
?-  retr_tup( flight, [ glasgow, dundee | _]).
```

If we wanted to make the relation *flight* transparent to Prolog programmers, we could define the procedure:

```
flight( From, To, Day, Dep, Arr) :-
    retr_tup( flight, [From, To, Day, Dep, Arr]).
```

By using a trap mechanism in MegaLog, it is possible to make all or specific external relations transparent to Prolog programmers. In subsequent examples, we assume this facility is used where the context makes it clear.

It should be pointed out that the efficiency of set evaluation of range queries can also be obtained in tuple at a time retrievals. For example, the query: Is there a flight from *London* to *Glasgow* on a *Monday* arriving before *10:00* hrs ? - can be efficiently formulated by passing the evaluation of the range condition to the underlying file manager system [23, 22, 24]:

```
?-  retr_tup( flight, [ london, glasgow, monday, Dep, Arr],
     Arr < 1000).
```

3.2 Deductive Relations

Here we generalize the above concepts. Say we want to create the deductive relation *connect/5*. This deductive relation is defined by declaring it:

```
?- connect <==> [ from, to, day, depart, arrive].
```

Notice the absence of data type specifications. Indeed, they are not required.

To populate a deductive relation, we insert clauses in it. Say that we wanted to state that there is a flight, once to the hour, starting at *8:30* hrs until *21:30* hrs every day, from *Frankfurt* to *Munich*:

```
?- insert_clause((connect(frankfurt,munich,Day,Dep,Arr) :-
      day(Day),
      period( 8:30, 1:00, 21:30, Dep),
      Arr is_time Dep + 0:50)).
```

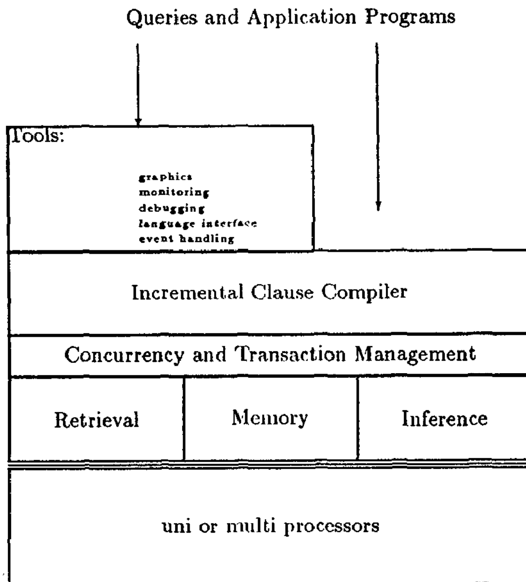
and to query it:

```
/* 'A flight from  Frankfurt to Munich on mondays ?' */
?-  connect( frankfurt, munich, monday, _, _).

/* 'Timetable of flights from Frankfurt to Munich' */
?-  frankfurt_munich isdr
      [ day, depart, arrive] :^: connect where
          from == frankfurt and to == munich.
```

4 Architecture

Users of modern information systems - in particular, users of current generation DBMS's - expect an interactive interface, and uninterrupted operation for very prolonged periods of time. To achieve these objectives, any system which uses compilation to achieve performance must rely on incremental compilation techniques and sophisticated garbage collection techniques. This, together with the requirements for portability, performance, persistency, shareability of knowledge, concurrency and a deductive capability, led us to design an abstract machine based on three major components: an *inference engine*, based on a derivative of the Warren Abstract Machine - WAM [25, 30, 32]; a *retrieval engine* built over the file manager system BANG [23, 22, 24]; and, a *main memory management subsystem* providing support for dynamic allocation of memory on demand, extensible and performing *full garbage collection* [8, 4, 14, 15, 16, 21]. The diagram below shows how these three components which form the kernel of the MegaLog system relate to each other. It also shows the other functional components of MegaLog's architecture:



Without entering into precise details, it should be noticed the integrating function performed by the memory manager. It avoids the replication of activities usually encountered in systems constructed by coupling or integrating two existing systems. The common memory manager not only makes possible the maintenance of good locality of reference, but it also eliminates the duplication of buffers. More importantly, by providing full garbage collection, it makes continuous operation for very long periods of time possible[4]. In addition, this garbage collection mechanism performs dynamic clustering of complex structures. We refer the reader to [4] for more details[3].

5 Compilation and Indexing

The principles of compilation used by MegaLog are in general the same as in a Prolog compiler, with two big exceptions: index generation and the use of associative instead of physical addresses in the code generated.

Base relations are directly stored in secondary storage and the algebra for manipulating them is interpreted. The overhead of interpretation in this context is minimal and hence there is no justification in performance terms to write a compiler for this sublanguage.

In contrast, due to the complexity of the terms allowed in deductive relations, the overhead of translation would be an unacceptable burden if interpretation were used. This was empirically demonstrated by our previous work on Educe [9]. Thus the deductive relations are populated by compilation of individual clauses, incrementally (although it is possible to process them in batches as well).

5.1 Incremental Compilation of Clauses

One of the key features of compiled Prolog programs is the relatively good selectivity of clauses within non-deterministic procedures. This has been achieved thanks to adequate indices constructed during the compilation of procedures. Unfortunately in the case of large logic knowledge bases, this is not realistic given the time it would take to do it. In an environment where procedures involving thousands of clauses can be shared and updated many times over relatively short periods of time, it would be unrealistic to work out new indices for the modified procedures each time one or more of the clauses in them are updated.

In MegaLog, procedures are dynamic and their clauses can be updated individually by concurrent processes, without causing a recompilation of whole procedures. A self adjusting (dynamic) form of indexing on persistent procedures is maintained. This persistent and dynamic form of indexing is provided by *BANG* [23, 22, 24], a grid type of file system capable of good response time on partial matching queries. See sub-section 5.2 below.

The compiler for MegaLog takes clauses which are compiled into code for the abstract inference machine. This code can be directly executed. However, it is more often the case that code is stripped of its physical references replacing them by associative references. For illustrative purposes, consider an arbitrary atom. This is represented in external code by a hash value that might even clash with the hash value for an entirely different atom. However, all references will be made physical again just before execution of a clause, leaving the run time loader to resolve the clash. These associative references indeed act as filters on retrieving clauses kept in persistent store.

5.2 Indexing

Once a clause is compiled, its object code - free of physical addresses - is stored in a heap file. The pointer to the compiled code, together with a symbolic representation of the arguments in the head of the clause, make up a descriptor for the clause. This symbolic representation is produced in identical fashion to the associative references, discussed above.

The descriptors for the clauses are maintained by the BANG file system. This dynamic management of indices makes possible the compilation of logic programs on the basis of one clause at a time, i.e. the incremental compilation of clauses (as opposed to full procedures). Thus updates are performed fast and without further consequences - as expected in a conventional data base management system. This point

[3] *Another technical report which precisely deal with this issue is in preparation*

should be stressed, since potential users of deductive systems might not be aware of the impossibility of providing this functionality in systems based on couplings or integrations of existing inference engines and commercial DBMSs.

6 Execution of Persistent Code

In this section we show how the compiled code, the index on descriptors and the relative addresses are put together for query evaluation. We present in some detail the tuple and set evaluation mechanisms used in queries over deductive relations. From this description the query evaluation strategy used over ordinary relations is obvious.

6.1 Tuple at a time evaluation

Tuple at a time evaluation is activated by a trap which activates the procedure *exec/1*. This procedure is made up of three parts: setting up, descriptor retrieval and execution.

In the first part, preparation for retrieval of descriptors is made. This work is only done once. From the user's query the relevant procedure, its arity and arguments are extracted. This information is used to determine the name of the relation that has the clause descriptors for the procedure. The list of arguments is used in addition to the name and arity of the procedure to build a basic frame from which alternative descriptors could be generated.

The second part uses the basic frame and the instantiation of the arguments in the query to generate alternative frames for retrieving and testing descriptors. These frames, one at a time, are used to search for clause identifiers in the relation holding the descriptors relevant to the query. Each clause identifier corresponds to a piece of code for a clause. All backtracking is implemented in this part.

The execution of the code takes place in the third and last part. The clause identifier is used to retrieve the code from the heap file. A request to the heap file manager loads the code and transforms the associative addresses into physical ones. The code is installed at an address selected by the heap file manager and executed. If the execution fails, backtracking takes place into the second part.

The use of associative addresses eliminates the need for garbage collection of the heap file, except for the minor task of recovering the space occupied by deleted clauses. This activity is almost negligible in time since it is triggered by the direct deletion of clauses and has no other consequences. In fact, memory recovery only takes place once all transactions involving the deleted clause are completed.

The memory used by the code during its execution is allocated by the heap file manager. The heap manager makes use of buffers and naturally is constantly re-using the same memory. This reduces I/O and improves locality of reference. It is obviously undesirable to have an excessive I/O traffic between main memory and disc, in particular if the number of bytes to move each time are few (which is typically the case for the code of individual clauses). Buffering by the heap manager reduces this traffic to a trickle. Locality of reference is already a serious problem of main memory Prolog systems. In particular the situation is aggravated by jumps amongst the different pieces of code for clauses within an indexed procedure. In MegaLog, the heap file manager avoids this by re-use of the same piece of memory, whenever possible at loading time.

The fact that execution is based on individual clauses, with no regard to the number of clauses making up the procedure, eliminates the need for garbage collection of a procedure table and/or a code heap in main memory. In MegaLog, the procedures kept in main memory (non persistent procedures) make use of a procedure table and of a main memory heap. Since these procedures are static, not much garbage is generated. However, even the garbage generated by the management of these procedures is collected.

6.2 Set evaluation

Set at a time evaluation is triggered by the execution of one of the operators *isdr/2* or *expand/2*. The most common of the derivatives is *expand/2*, used to generate the extension of a deductive relation. The algebra expression on the right of *isdr* is interpreted over the relations containing the descriptors. The descriptors are of a fixed size for each derived relation and hence particularly suited for manipulation by the *BANG* file manager. This file manager is particularly efficient in handling partial match type of queries: precisely the type of queries being posed by the algebra of descriptors. An abbreviated form of the algorithm to evaluate the algebra expression is as follows:

1. Transform the expression over the deductive relations to an algebra expression over the associated base relations containing the descriptors.
2. Evaluate the algebra expression over the base relations containing the descriptors. This generates an intermediate base relation T, which contains the relative addresses in the heap file of the relevant code. This is in fact a filtering action.
3. From the original algebra expression, generate a new expression to be used for tuple at a time testing. Refer to this expression as C.
4. For each tuple in the temporary relation T, retrieve the code P referred to in the tuple.
5. If the operator is *isdr* then test that at least one answer can be generated from this code (P). To do this execute the code P and the checks C, in an interpretive manner. For each tuple contributing to the answer, generate a new clause from the original clause associated to the code P and the checks C used during the current testing. Compile this new clause and store it in a new deductive relation.
6. If the operator is *expand*, proceed similarly to the above case but, instead of generating the linking clause, generate all the facts associated with the tuple in the intermediate relation. Compile these facts and store them in the new persistent relation.
7. Finally, remove the intermediate relation T.

By use of the strategy described, the performance of set at a time evaluation is almost entirely dependent on the evaluation capabilities of the underlying engine for the relational algebra of ordinary relations. Execution of compiled code is delayed to the last stage where it is only done for verification purposes.

7 Some Results

Initial tests have been performed on our airline knowledge base. We have experimented with procedures having as many as a thousand clauses (rules as opposed to facts). If these clauses were flattened to the corresponding facts they would generate some fifteen thousand tuples of complex terms, including structures. The initial tests show that response time for tuple at a time evaluation on clauses kept in persistent store appears similar to its main memory counterpart. Of course, if the same evaluation technique were used for universally quantified queries, then the equivalent would not hold. For set evaluation of derived relations in the case of queries involving one relation of some thousand clauses, the time to perform any of the algebra operations is nearly always under half a second. For more complex queries, such as a join over the same relation with a pair of selection conditions, e.g.: *'produce the timetable of all the flights from Munich to London with one change of plane in between'* - the time is about one second[4].

[4]Sun/3, 8 Mbytes of RAM and a file server as external store.

Tests on more conventional type of applications have been reported in [14]. Strictly within the functionality of commercially available DBMSs, MegaLog performs similarly to them. Used in an environment where the required indices can be predicted beforehand, it can often outperform the best commercially available relational DBMSs. The more so, in the cases in which the shape of queries to relations cannot be predicted (precisely the deductive case). It also outperforms them in queries involving partial match retrievals where the indices used by the DBMS are not optimal to the problem. It emphatically beats them in the case where the queries refer to attributes/arguments with no indices. This last case is a very frequent occurrence in a deductive environment (unpredictable queries are the norm rather than the exception).

8 Conclusions and further work

We have presented here a description of the design and implementation of the MegaLog system. The current implementation as it stands provides continuous operation, persistency, dynamic updates of clauses, transactions and concurrency for managing clauses in a multi-user environment[5]. A more generic form of query evaluation is provided by use of unification instead of (conventional) retrieval over a persistent store. The key issue of performance is resolved by the use of clause compilation and a suitable file management acting harmoniously with the execution engine.

The evaluation techniques for queries over a logic knowledge base are original and they eliminate the need for garbage collection of the persistent store. High levels of performance operating over the external store are supported at run time by the filtering done by the descriptors. Efficient management of the descriptors is obtained by the use of the *BANG* file system and the algebra implemented on top of it.

Other complementary features of the MegaLog system, are: a window debugger, graphics capabilities, monitoring tools, shared memory management and the transactions subsystem.

Future work includes the development of applications to explore the frontiers of the technology under discussion. In particular, it is necessary to discover what is the size and complexity of knowledge systems that the technology can efficiently support. This work is of course not separated from the general question of optimization techniques for logic knowledge base systems.

We are confident that our basic objectives have to a large extent been achieved in MegaLog. Deductive languages for data bases such as LDL [19] can be ported to it without serious problems. In fact, we have implemented the SALAD [19] application of LDL on MegaLog in a very short time, obtaining good performance. We have also used MegaLog in conventional relational applications with very good results. Regarding object oriented data base systems it is still too early to report results. However, we know of at least two projects using MegaLog to implement such systems. Future work will almost certainly be centred on providing more and better facilities for this latter purpose.

Acknowledgements. Members of the MegaLog team - past and present, have spent several years in transforming the concepts presented here into reality. Particularly significant contributions to the implementation of the system were made by Michael Dahmen, Philip Pearson, Geoffrey Macartney and Peter Bailey. Mike Freeston has also contributed enormously with his work on BANG and in discussions directly related to this work. More recently, Luis Hermosilla has contributed in benchmarking and tuning the system. Many others have also contributed by providing graphics and applications, and by testing and debugging.

[5]For obvious reasons of space, many of these features are not described in this paper.

References

[1] M. Atkinson, P. Bailey, K. Chisholm, P. Cockshot, and R. Morrison. PS-Algol: A language for Persistent Programming. In *10th Australian National Computer Conference*, pages 70–79, Melbourne, September 1983.

[2] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In Carlo Zaniolo, editor, *Proc. of the ACM-SIGMOD Conf. on Management of Data*, pages 16–52, Washington, D.C., USA, May 1986.

[3] H. Boas, P. Boas, and C. Doedens. Extending a Relational Database with Logic Programming Facilities. Technical report, IBM INS-Development Center - The Netherlands, 1984.

[4] J. Bocca and P. Bailey. Logic Languages and Relational DBMSs - The point of convergence. In M. Atkinson, P. Buneman, and M. Morrison, editors, *Proc. Appin II Workshop on Persistent Object Stores*, pages 346–362, Computing Sc. Department - Glasgow University, UK, August 1987.

[5] J. Bocca, M. Dahmen, M. Freeston, G. Macartney, and P. Pearson. KB-PROLOG, A Prolog for Very Large Knowledge Bases. In *Proceedings of the Seventh British National Conference on Databases (BNCOD-7)*, Edinburgh, U.K., July 1989.

[6] J. Bocca, M. Dahmen, G. Macartney, and P. Pearson. Kb-prolog user manual. Technical Report KB-31, ECRC, April 89.

[7] J. Bocca, H. Decker, J-M. Nicolas, L. Vieille, and M. Wallace. Some steps towards a DBMS based KBMS. In H-J. Kugler, editor, *Proc. 10th World Computer Congress*, Dublin, Ireland, September 1986. IFIP.

[8] J. Bocca, M. Meier, and Villeneuve D. The specification of a compiler with high performance and funtionality - SEPIA Prolog. Technical Report IR-PC-1, ECRC, May 1987.

[9] J. Bocca and P. Pearson. On Prolog-DBMS Connections: A Step Forward from EDUCE. In P. Gray and R. Lucas, editors, *Proc. Workshop on Prolog and Data Bases*, Coventry, England, December 1987.

[10] Jorge Bocca. EDUCE - A Marriage of Convenience: Prolog and a Relational DBMS. In R. Keller, editor, *Proc. '86 SLP Third IEEE Symposium on Logic Programming*, Salt Lake City, Utah, USA, September 1986. IEEE.

[11] Jorge Bocca. EDUCE - User Manual. Technical Report Internal KB Report, ECRC, 1986.

[12] Jorge Bocca. On the Evaluation Strategy of EDUCE. In Carlo Zaniolo, editor, *Proc. 1986 ACM-SIGMOD International Conf. on Management of Data*, Washington, D.C., USA, May 1986. ACM.

[13] Jorge Bocca. - Educe* - A logic programming system for implementing KBMS's. In *Proceedings of the Seventh British National Conference on Databases (BNCOD-7)*, Edinburgh, U.K., July 1989.

[14] Jorge Bocca. Compilation of Logic Programs to Implement Very Large Knowledge Base Systems - A Case Study: Educe*. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, California, USA, February 1990. IEEE.

[15] Maurice Bruynooghe. The memory management of Prolog implementations. In *Logic Programming*, pages 83–98, 1982.

[16] Maurice Bruynooghe. A note on garbage collection in Prolog interpreters. In *Proceedings of the First International Logic Programming Conference*, pages 52–55, Marseille, September 1982.

[17] C. L. Chang and A. Walker. PROSQL: A PROLOG Programming Interface with SQL/DS. In *Proc. of the First Int. Workshop on Expert Database Systems*, Kiawah Island, South Caroline, USA, October 1984.

[18] W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–162, 1990.

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

[20] O. Deux et al. The Story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, 1990.

[21] Tick Evan. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, 101 Philip Drive, Assinippi Park, Norwell, Massachusetts 02061, USA, 1988.

[22] Mike Freeston. Grid files for efficient Prolog clause access. In P. Gray and R. Lucas, editors, *Proc. Workshop on Prolog and Data Bases*, Coventry, England, December 1987.

[23] Mike Freeston. The BANG File: A New Kind of Grid File. In U. Dayal and I. Traiger, editors, *Proc. 1987 ACM-SIGMOD International Conf. on Management of Data*, San Francisco, USA, May 1987. ACM.

[24] Mike Freeston. Advances on the design of the BANG File. In *3rd International Conference on Foundations of Data Organization and Algoritms (FODO)*, Paris, France, June 1989.

[25] J. Gabriel, T. Lindholm, E. L. Lusk, and R. A. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Technical Report ANL-84-84, Argonne National Laboratory, 1984.

[26] Y.E. Ioannidis, J. Chen, M.A. Friedman, and M.M Tsangaris. BERMUDA - An Architectural Perspective on Interfacing Prolog to a Database Machine. In L. Kerschberg, editor, *Proc. of the 2nd International Conference on Expert Database Systems*, pages 91–106, Tysons Corner, Virginia, USA, April 1988.

[27] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–124, 1990.

[28] E. Sciore and D. S. Warren. Towards an integrated Database-Prolog system. In *Proceedings First International Workshop on Expert Database Systems*, pages 801–815, Kiawah Island, South Carolina, USA, October 1984.

[29] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, 1990.

[30] H. Touati and A. Despain. An empirical study of the Warren Abstract Machine. In *Proc. Symposium on Logic Programming '87*, pages 114–124, San Francisco - USA, September 1987.

[31] Y. Vassiliou, J. Clifford, and M. Jarke. Access to specific declarative Knowledge by Expert Systems: The impact of Logic Programming. *Decision Support Systems*, 1(1), 1984.

[32] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Report tn309, SRI, October 1983.

[33] David H. D. Warren. Logic Programming and Knowledge Bases. In *Proc. of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, pages 69–72, Islamorada, Florida, USA, February 1985.

[34] K. Wilkinson, P. Lyngboeck, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):63–75, 1990.

[35] Carlo Zaniolo. Object-Oriented programming in Prolog. In *Proc. International Symposium on Logic Programming*, pages 265–270, Atlanta City, N. J., February 1984.