Rules for Implementing Very Large Knowledge Base Systems

Jorge B. Bocca
Johann Christoph Freytag

European Computer-Industry Research Centre
Arabellastr. 17
D-8000 München 81, Federal Republic of Germany

{jorge,jcf}%ecrcvax.uucp@unido.EDU

Abstract

With the current research effort in knowledge base management systems (KBMSs) the interest in storing, retrieving, and processing rules efficiently, and using them for describing and implementing such systems has increased significantly. In this paper we contribute to this subject in two ways. First, we introduce **Educe***, an integrated programming system for building large knowledge base systems. In particular, we describe the treatment of rules and complex logic terms, examine at the physical level the compilation of logic programs and its subsequent storage, retrieval and execution in the specified context, and discuss performance related aspects of our system.

Second, we provide an overview on how to use rules for organizing the processing of user requests in a database/knowledge base environment. Specifically, we discuss aspects of rule-based query optimization and compilation. Finally, we outline additional query processing problems that could benefit from a rule-based specification and a rule-driven implementation.

1 Introduction

With the current research effort in knowledge base and extensible database management systems the interest in using rules for describing and implementing such systems has increased significantly. When considering knowledge base systems that are based on the emerging ideas of deductive databases and logic programming using rules for specification and implementation purposes becomes even more attractive. However, a serious drawback appears when the complexity and size of the applications considered goes beyond the scope of small research prototypes. When scaling up systems that are based on existing technologies on a "stand alone" basis or with some loose integration, these systems have failed to produce the performance required by such applications.

In this article, we present our views and the general lines of our research work on how significant improvements in performance could be attained. Our work approaches the problem from two different but nevertheless convergent perspectives. At the physical level, we are studying the handling of logic programs in compiled form by an external data base (EDS); while at the logical level, we search for performance improvements by means of the use of transformation rules.

In theory at least the integration of programming languages based in the logic paradigm, e.g. Prolog, and relational DBMSs appears as one of the most promising technologies to be used for constructing Knowledge Base Systems. Relational systems have reached a considerable acceptance and their conceptual basis is generally recognized as simple and powerful. Logic programming has also demonstrated its relative merits in the field of artificial intelligence, in particular in the development of rule based expert systems.

Also, given their congenial theoretical foundations, they seem in principle, particularly suited to provide the ground on which development systems for constructing Very Large Knowledge Base Systems could be implemented. However, in terms of

implementing the integration of a relational system to a programming language such as Prolog, some significant performance problems appear. This is largely due to the extra complexity introduced by operations such as recursion, unification, and resolution over data held and controlled by the relational DBMS. In particular, performance becomes so poor whenever rules are involved in queries and/or transactions. Typically, in cases where a naive interface between the (logic) deductive and the relational engines is used, applications beyond the trivial cases cannot be run.

Loose coupling and tight integration of a Prolog interpreter and a relational DBMS was tried by us in Educe [Boc86b] [Boc86a]. The power of this system, was demonstrated in practical terms with the implementation of the prototype Knowledge Base System - KB2 [Wal86]. As expected, in the retrieval of facts the performance of Educe is satisfactory even in reasonably large relations; however, the retrieval of rules and their consequent usage can lead to serious performance degradation.

In the light of the above arguments, in this article we discuss the integration of logic programming and relational technologies from their principles, but with the emphasis in performance. For this, we have selected for discussion what we believe is at the center of a high performance system based on the integration of the logic programming and relational technologies. These are: compilation in the EDB, management of control and transformation rules for optimization. For the purpose of illustration, we use **Educe*** [Boc89] - a system whose architecture precisely rest on the principles discussed here.

2 The Case for Compilation

Our analysis of performance related issues in Educe established that three factors contribute rather badly to performance whenever rules are stored in the external data base (EDB): the use of an interpreter instead of a compiler, poor selectivity on the retrieval of rules stored in the External DB and frequent assertion (and erasure) in main memory of rules stored in the External DB. All of these three factors point towards the usage of some form of compiled code to be stored for rules in the external relational storage.

Performance improvements of compilation over interpretation of logic programming languages have been clearly shown with the appearance of Prolog compilers. It is not unusual to have performance increased by several order of magnitude when moving from an interpreter to a compiler.

Rules in Educe are stored in the EDB in source form. In order to use them, they have to be retrieved from the EDB, parsed and asserted in main memory. To improve the selectivity on retrievals, Educe stores some extra information. In a system based on compilation it is possible to use the compiler itself to collect the additional information to be used to improve the selectivity of retrievals.

Asserted clauses in main memory, whether rules or facts are very expensive to use, in terms of cpu time. Nevertheless, in a system that stores rules in the EDB in source form it is an unavoidable operation.

It should also be noticed that rules kept in source form in the EDB not only need to be searched for in the EDB and asserted in main memory, prior to execution, but they also need to be erased to make room for the next set of rules to come from the EDB. Potentially, a given rule can be asserted in and erased from main memory, many times during a session. Compiled code stored externally makes these assertions and erasures unnecessary. In addition, the very time consuming activity of parsing general logic terms would not be required at all when loading from the external DBMS.

It is on the strength of these arguments that we decided to use compiled logic clauses in the external relational store. This technique was adopted in the **Educe*** system. In particular, in the case of complex compound terms and rules, they are practically always maintained by **Educe*** in compiled form in the EDB.

3 The Fundamentals of an Architecture

Even from their most primary conceptions, high performing engines for relational systems and for logic deductive systems are set to counteract with each other when put together without careful analysis of their respective purpose and function. A suitable strategy of integration should attempt to do some form of packing to transfer code from disc to memory while still executing the code one term at a time. The validity of this strategy is confirmed by our tests which show that the time split between I/O and cpu usage [BP87] is very different to the one in relational

systems. In our case and much more frequently than in relational systems, the computation of answer to queries becomes cpu bound. Much more so, in the case of rules.

To implement the above strategy in the kernel of **Educe*** three major areas of the interaction were subject to detailed study: representation of facts and rules in the EDB, the management of control flow and the management of memory. From these studies the architecture for **Educe*** was derived. This is made up of four major components:

1. An incremental compiler which produces code for **Educe***'s internal virtual machine, except that memory addresses are associative instead of the normal absolute ones.

2. A dynamic loader. This loader, at run time, resolves associative addresses, adds procedural and other forms of control code to the clausal code stored in the EDB. This makes the retrieved code runable in **Educe***'s virtual machine.

3. A very fast emulator. This emulator is derived from the Warren Abstract Machine (WAM), the most popular emulator for compiled Prolog programs [War83]. This is the most efficient and powerful model used nowadays for the construction of Prolog compilers. The emulator interprets the code delivered by the dynamic loader.

4. The BANG [Fre87a] file system. This is a grid type file particularly suited for efficient Prolog clause access. This file system has been developed by M. Freeston, within ECRC. Although, BANG is as a separate project to our, it has been developed in parallel to the Educe/**Educe*** projects, in mutually collaborative terms. For details of it, we refer readers to [Fre87a].

4 Control

Because of its relative importance to performance, we would like to close the discussion of the physical level with an insight into the implementation of control features in **Educe***.

4.1 Choice Points

Backtracking is implemented in the Warren Abstract Machine (WAM) by the use of **choice points**, a mechanism to record a particular state in the computation so that in the case of a failure a return to this state becomes possible and hence, a new alternative computation could be attempted.

In a system that integrates the inference and the relational engines, it makes sense to use the control features of the inference engine, since the conventional relational engine lacks the control features for the implementation of backtracking. Thus, a better way of implementing the integration is to extend the logic deductive language with deterministic procedures to interface with the low level record manager of the relational DBMS. Precisely this was done in the integration of the relational and the deductive components in Educe.

However a problem still remains to be solved, that is how backtracking is implemented, if Prolog is used. The *repeat* predicate creates the choice point necessary for backtracking to take place. Thus, blindly and without regard to the nature of the query or characteristics of the base relation involved, a choice point is created for each query on a base relation. The significance of this should not escape us. Empirical studies of the WAM [TD87] have asserted that choice point references are the *single most significant contributor to the total number of data references.*

It might seem unavoidable to have to set up the choice points when dealing with relations in the EDB, but this is not always the case. There are important exceptions and some of them show up when dealing with rules stored externally in the EDB.

Rules in the EDB are supported in Educe by means of an exception handling mechanism in the host Prolog system. At the center of the implementation there is an interpreter program that is trapped when no predicate is found in main memory to evaluate a given query.

In the case of rules, the interpreter retrieves all the clauses for the procedure which matches the *Goal*. This is needed to freeze the definition of the procedure, thus avoiding possible inconsistencies introduced by updates of the procedure while still in use. Performance is badly affected by the poor selectivity of this policy.

The elimination of unnecessary choice points is achieved by the use of a deterministic procedure to collect all the clauses for the wanted predicate, at once. This has also the desirable characteristic of grouping related code together for transfers and/or other processing. The procedure uses for

this purpose information normally maintained by the schema of the EDB, such as primary keys and secondary indices. This mechanism filters out unsuited clauses and in some cases, avoid the creation of choice points for these non-deterministic procedures.

4.2 Indices

Since compilation to produce instructions for the Warren Abstract Machine (WAM) is geared towards evaluation of one term at a time, it makes good sense to index always in the first argument[1] and according to data type and value. This effectively satisfies the desirable effect of early identification of failures. Of course, it also acts as a filter to reduce the number of clauses to inspect in a given procedure, i.e. it does similar work to a secondary index mechanism as used in relational systems. However, the results are considerably poorer than those obtainable with a good index mechanism in relational systems. The reason for this is that normally only one argument is used to construct the index. The alternative of using more than one argument in the construction of the key has the tendency to increase the size of the code generated in an exponential manner. Code specific to each of the alternatives has to be generated.

The above points have to be considered without minimizing the importance to performance of indexing in the WAM. For example, it should be noticed that indexing in the WAM, often transforms a non-deterministic procedure into a number of purely deterministic procedures (or at least some of them become deterministic). This avoids trying unfruitful alternatives, or put in another way, eliminates the need to create choice points.

A feature of no value to relational DBMSs - indexing over the type of the term - is very effective in an inferential engine. Its obvious application is in filtering applicable clauses of a given rule kept in the EDB. This form of indexing increases the size of the generated code in a more manageable fashion, since the possibilities are fewer.

In **Educe*** indexing on type and value is supported. The indexing mechanism of the EDS is used to filter the code for relevant clauses and code to support indexing in main memory is added by the dynamic loader, together with other control code.

[1] *attribute* in relational terminology

4.3 Unification

The storage of compound terms and in particular of rules in source form in the EDB does not only represents a penalty in performance due to parsing and assertion, but it also inhibits the use of more elaborated filters to be used in retrieval operations. For example, if in the relation **stock**(*product_no, quantity*), we have the following tuples:

```
stock( 21, kilos(345)).
stock( 78, litres(210)).
stock( 95, kilos(1234)).
stock( 12, litres(345)).
stock( 26, kilos(1247)).
```

In order to answer the query *"Give the number and quantity of the products measured in kilos"*

```
?- stock( X, kilos(Y)).
```

based on Educe's principles, i.e. integration of the deductive and relational components in a loose form, we would (normally) need to retrieve from the EDB into main memory, all the tuples in the relation **stock** and only then proceed to do unification.

To eliminate this form of performance penalty, **Educe*** does pre-unification in the EDS. This is implemented by providing the EDS with a pre-unification unit that is used on the code with relative addresses maintained by it.

5 Using Rules for Query Processing

The ability to store and retrieve rules efficiently leads to the obvious idea to organize various processing steps by rules in a KBMS. In particular, the processing of user requests against the knowledge base, especially their optimization, is a perfect candidate for such an effort. This coincides with the intend of extensible DBMSs to organize query processing steps in such a way that they can be adapted to the varying requirements of different application areas. In the sequel we briefly outline how rules can support the processing of user requests in a flexible and efficient manner. Although this work has been carried out mostly in the context of relational DBMSs we believe that the techniques developed and the results obtained are equally applicable in a KBMS processing environment.

5.1 Rules for Query Optimization

During the processing of user requests the query optimizer is responsible for generating an optimal query evaluation plan (QEP), i.e. the optimizer selects the best QEP among different alternatives. In [Fre87b] we define sets of transformation rules that specify how to derive different QEPs in a defined target language - for our purpose a relational algebra-like language - from a query expressed in some user language. The sets of transformation rules generate the algebraic query form, produce alternatives how to access the different relations (i.e. sequential search, index access etc.), and explore alternative join orders and join methods. These transformation rules can easily be extended to generate more complex QEPs, i.e. accessing multiple indexes, storing intermediate results in intermediate relations for later reuse, or extending QEPs for evaluating queries in a distributed processing environment.

In [Loh88] Lohman extends and refines the set of transformation rules for an extensible query optimizer in the Starburst project at the IBM Almaden Research Center; the details of the optimizer implementation are described in [LFL88]. The implementation demonstrates that the rules can smoothly be merged with the search strategy and the cost functions, the other two other major aspects of query optimization. Candidates for search strategies are the greedy search, breadth-first search, k-step look-ahead search, or the search by simulated annealing. Cost functions are necessary to compare QEPs to determine which one among different choices will perform best during the evaluation of the query.

As a first step to integrate results of query optimization developed for relational DBMSs, into a KBMS environment we have prototyped a simplified, rule-based query optimizer in Prolog with several search strategies and an extending set of transformation rules [Fre89]. This optimizer prototype is based on a state-based optimization model that leads to the implementation of well defined subcomponents by clearly showing the interaction between transformation rules, search strategies and cost functions [Fre89].

5.2 Rules for Translating Queries into Iterative Programs

QEPs that are generated by the query optimizer generally express the different evaluation steps still in a set-oriented manner. On the other hand, low-level interfaces often offer only a tuple-at-a-time interface for an efficient evaluation. To bridge the gap between set-oriented QEPs and the tuple-at-a-time interface an additional transformation step becomes necessary. In [FG86a] and [FG89] we describe an algorithm that transforms QEPs with various algebraic operators into iterative programs based on recursive function definitions for QEP operators. To perform the desired transformation this algorithm incorporates four classical transformation steps that are well known in the area of program transformation:

1. **Unfolding:** Replace a function call by its definition.

2. **Simplification:** Exhaustively apply a set of transformation rules to simplify a given expression.

3. **Folding:** The reverse of unfolding, i.e. replace an expression by the corresponding function call if that expression resembles the function's definition.

4. **Replacing Recursion by Iteration:** If possible, replace the recursive program by an equivalent iterative one for performance reasons.

The algorithm for QEPs contrasts with a transformation algorithm for arbitrary general programs in that the former does not need any user intervention for a successful application of these four transformation steps while the latter does. Our algorithm can easily be extended by adding new QEP operators and the corresponding recursive function definitions without changing its logic [FG89].

In [FG86b] we propose algorithms that rely on the functional programming paradigm [Bac78] as the underlying formalism for transforming aggregate queries, i.e. queries containing aggregate functions such as *count* or *average*, into iterative programs. We introduced two independent transformation steps both of which are easily expressible by a small set of transformation rules: *horizontal transformation* uses the functional programming notation to generate the desired program structure, *vertical transformation* manipulates recursive function definitions to produce the final iterative programs. The latter transformation is especially interesting since the algorithm can produce two alternative iterative

programs that exhibit different computational behaviors. Both transformations might be desirable for efficiency reasons [FG86b].

5.3 More about Query Processing by Rules

To show that other problems in the area of database and knowledge base systems can benefit from the concept of rules we mention two other problems that we currently work on in the context of knowledge bases and which also seem to be amenable to transformation rules.

When including object-oriented concepts into database or knowledge base systems, it seems to be advantageous to represent objects internally by tuples stored in a relation or in a set of relations. The choice of implementing object concepts by relational concepts allows us to store data and to evaluate queries using existing, well understood relational technology. However, there does not seem to exist one agreed, unique mapping from the object to the relational level. One could define this mapping by transformation rules, thus providing a flexible specification for possibly changing the mapping over time. By applying meta-compilation techniques as outlined in [FMW88] one can produce a object-relation compiler that is equally efficient as compiler that is hand-coded for one possible mapping.

When extending relational DBMSs to KBMSs the task of "logical" optimization, i.e. optimization that includes techniques such as semantic query optimization or query simplification, becomes increasingly important, but also increasingly more complex. To adapt the amount and the kind of optimization to varying requirements it is desirable to organize the different optimization steps in a flexible manner. We currently experiment with the notion of *strategy rules* as a way to achieve the desired flexibility. Ullman already introduced strategy rules in [Ull85] to allow different transformations of recursive queries depending on their syntactical structure. It looks promising to us to use the same idea for organizing the complete process of query optimization.

6 Conclusions

In this article, we examined one of the most promising technologies being used to develop the next generation of Knowledge Base Management Systems and Intelligent Data Bases. This technology is based on the integration of two already existing technologies: logic programming and relational systems.

We identified poor performance as the major problem to be solved before the integration approach could achieve successful results. In order to achieve the required performance gains, we adopted as general strategy, a very tight integration of technological principles. This approach was adopted after our own experiences with Educe - a system based on coupling and integration of two existing system.

The architecture of **Educe*** - a new system to advance on the work on Educe, seeks to reduce the inefficiencies caused by a rather loose interface of the two components: the deductive and the relational.

In **Educe***, the performance deterioration caused by frequent parsing of rules, their assertions in and erasures from main memory is eliminated by the use of compiled code in the EDB. However, because of persistence of code in the EDB and the need to garbage collect within a given session, only relative addresses could be generated for the code in the EDB. A dynamic loader is activated to map relative addresses into physical addresses. This loader also adds the needed control information to the code prior to execution.

Coming from the database side we introduced rules as a means to specify and implement the processing of user requests to improve their evaluation on the knowledge base. The discussion of query optimization and query translation shows that the rule-driven processing is especially attractive in an environment such as **Educe*** that provides the prerequisite for such an approach. We hope to gain more experience in using rules and can extend the current algorithms and techniques for organizing the processing of user requests during a future design and implementation of an integrated KBMS.

7 Acknowledgements

We would like to thank fellow members of the Knowledge Base group at ECRC for their contributions in discussions relating to this work.

References

[Bac78] J. Backus. Can Programming be liberated from the von Neuman Style ? A Functional Style and its Algebra of Programs.

Communications of the ACM, 21(8):613–641, August 1978.

[Boc86a] J. Bocca. EDUCE - A Marriage of Convenience: Prolog and a Relational DBMS. In *Proceedings '86 SLP Third IEEE Symposium on Logic Programming*, Salt Lake City, Utah, USA, September 1986.

[Boc86b] J. Bocca. On the evaluation strategy of educe. In *Proceedings ACM SIGMOD 1986, Washington, D.C.*, May 1986.

[Boc89] J. 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.

[BP87] J. Bocca and P. Pearson. On Prolog-DBMS Connections: A Step Forward from EDUCE. In P. Gray and R. Lucas, editors, *Proceedingsof the Workshop on Prolog and Data Bases*, Coventry, England, August 1987.

[FG86a] J.C. Freytag and N. Goodman. Rule-Based Translation of Relational Queries into Iterative Programs. In *Proceedings ACM SIGMOD 1986, Washington, D.C.*, pages 206–214, May 1986.

[FG86b] J.C. Freytag and N. Goodman. Translating Aggregate Queries into Iterative Programs. In *Proceedings VLDB 1986, Kyoto, Japan*, pages 138–148, August 1986.

[FG89] J.C. Freytag and N. Goodman. On the Translation of Relational Queries into Itarative Programs. *ACM Transactions on Database Systems*, 14(1), March 1989.

[FMW88] J. C. Freytag, R. Manthey, and M. Wallace. Mapping Object-Oriented Concepts into Relational Concepts by Meta-Compilation in a Logic Programming Environment. In *Proceedings of the Second International Workshop on Object-Oriented Database Systems, Bad Ebernburg*, Lecture Notes in Computer Science, pages 204–208. Springer Verlag, September 1988.

[Fre87a] M. Freeston. Grid files for efficient Pprolog clause access. In P. Gray and R. Lucas, editors, *Workshop on Prolog and Data Bases*, Coventry, England, August 1987.

[Fre87b] J.C. Freytag. A Rule-Based View of Query Optimization. In *Proceedings ACM SIGMOD 1987, San Francisco, CA*, pages 173–180, May 1987.

[Fre89] J.C. Freytag. The Basic Principles of Query Optimization in a Relational Database Management System. In *Proceedings IFIP 1989, San Francisco, CA*, August 1989.

[LFL88] M.K Lee, J.C. Freytag, and G. Lohman. Implementing an Interpreter for Functional Rules in a Query Optimizer. In *Proceedings VLDB 1988, Los Angeles, CA*, pages 218–229, August 1988.

[Loh88] G. Lohman. Grammar-like Functional Rules for Representing Query Optimization Alternatives. In *Proceedings ACM SIGMOD 1988, Chicago, IL*, pages 18–27, June 1988.

[TD87] H. Touati and A. Despain. An empirical study of the warren abstract machine. In *Symposium on Logic Programming '87*, pages 114–124, San Francisco - USA, September 1987.

[Ull85] J. D. Ullmann. Implementation of logical Query Languages for Databases. *ACM Trans. on Database Systems*, 10(3):289–321, September 1985.

[Wal86] M. Wallace. Kb2: A knowledge based system embedded in prolog. Technical Report TR-KB-12, European Computer-Industry Research Centre, Munich, August 1986.

[War83] D.H.D. Warren. An abstract prolog instruction set. Technical Report tn309, SRI, October 1983.