Compilation of Logic Programs to Implement Very Large Knowledge Base Systems - A Case Study: Educe*

Jorge Bocca

ECRC, Arabellastr. 17, D-8000 München 81, West Germany
jorge@eccovax@pyramid.pyramid.com

Abstract

One of the most promising technologies being used for constructing Knowledge Base Systems is the integration of programming languages based on the logic paradigm, e.g. Prolog, and relational DBMSs. In theory at least it provides a base on which development systems for constructing Very Large Knowledge Base Systems could be implemented. However, to achieve this goal in practical terms, improvements in performance by several orders of magnitude over existing systems must be obtained. Towards this performance improvement we examine the compilation of logic programs and their subsequent storage, retrieval and execution in the specified context. We inspect the relationship between relational engines as found in relational DBMSs and the Warren Abstract Machine (WAM) [22], the most efficient and powerful model used for the construction of Prolog compilers. The development of the Educe* system is used to illustrate implementational issues.

1. Introduction

Although relational systems have achieved a considerable acceptance and their conceptual basis is generally recognized as simple and powerful, in terms of implementation they still appear to present some performance problems. When they are front-ended to languages based on logic, e.g. Prolog, the problem is made worse because of the extra complexity introduced by new operations such as recursion, unification, resolution, etc.

The sort of loose coupling and tight integration of a Prolog interpreter and a relational DBMS as used in Educe [4, 5] makes sense because of their congenial theoretical basis. The power of a system constructed by the union of the two technologies was demonstrated in practical terms with the implementation of the prototype Knowledge Base System - KB2 [21]. The drawback however is in poor performance. Although on the retrieval of facts the performance of Educe is satisfactory even in reasonably large relations, retrieval of rules and their consequent usage can lead to major performance problems. Towards a solution of the performance problem we consider how to maintain logic programs to be maintained in compiled form in an external relational store.

Through an analysis of the current technology of relational systems and compilation of logic programming, we derive the basic elements used in the design of the inference engine for Educe*. This is a system for constructing Knowledge Base Management Systems (KBMS) based on the co-existence of the logic model used by languages such as Prolog, and the relational model of data. The union is one of principles rather than of component parts, and it has been enriched by extensions to the syntax and semantic of both types of systems. On the relational side, unification on secondary storage is supported, instead of the more restricted retrieval facilities found in conventional relational DBMSs. Manipulation of large data sets and new forms of evaluation appear as extensions of the language Prolog (supported as a subset of facilities). A stronger typed language in a fluid combination of strategies of evaluation are put together in Educe* to achieve an unobtrusive syntax, capable of giving a power of expression beyond the one offered singly or in combined form by the other two types of system. In essence, Educe* provides a powerful environment in which to implement next generation Knowledge Bases.

However, although facilities are provided in Educe* to help, for example, in the detection of cyclic data, no attempt is made to define the 'best' strategy to handle recursion or the halting problem [1, 2, 11, 20]. For a more detailed description of the data model in Educe*, we refer the reader to [5]. In this paper we concentrate on the management of rules and complex compound terms stored in the External Data Base (EDB).

2. A Comparison of Principles of Operation

Our studies of performance related issues in Educe showed that three factors contribute to poor performance whenever rules are stored in the 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 [7]. All three factors point towards the usage of some form of compiled code to be stored for rules in the external relational storage, as shown below:

1. Performance differences between compiled and interpreted logic languages have been clearly demonstrated with the appearance of Prolog compilers. It is not unusual to have performance increased by several orders of magnitude when moving from an interpreter to a compiler. However, in the context of a system that integrates Prolog and a Relational DBMS, compilation into native code is not possible due to the need for garbage collection of externally stored compiled code. Nevertheless, the possibility of storing code for a virtual machine extended to handle associative addresses is still attractive in performance terms.

2. The obvious way of storing rules in external storage in Educe was in source form. In addition, Educe stored some extra information to make rule selection more selective. 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.

3. The use of asserted clauses in Prolog, whether rules or facts, is very expensive in terms of cpu time. But in a system that stores rules in the EDB in source form it is an unavoidable operation. In order to use rules kept in the EDB, they have to be searched for in the EDB, asserted, executed and finally erased to make room for the next rule(s) coming from the EDB. Notice that potentially a given rule can be asserted and erased thousands of times in main memory. Compiled code stored externally makes these assertions and erasures unnecessary and also avoids the very time consuming activity of parsing general logic terms when loading from the external DBMS.
Tests performed with Educe and KB-2 strengthened the above arguments and convinced us of the need to maintain rules in compiled form in the EDB. But for this, a model on which to base the compilation process became a new necessity. Hence, the need to study in some depth the relationship between relational engines as found in relational DBMSs, and the compilation of logic languages such as Prolog.

Prolog has established itself as the most important of languages for logic programming, and the fastest compilers for conventional hardware are based on the Warren Abstract Machine (WAM) model [23]. This makes the WAM the prime candidate for the compilation model of the successor to the Educe system: KB-Prolog [8] and Educe+ [9]. It should be noticed that most if not all of the other major models used for the compilation of Prolog are in essence very similar to WAM [10, 18]. In addition, the WAM is not restricted to Prolog compilation since it has been proved as a very efficient model in compilers for other logic languages and/or extensions to Prolog [15, 16].

Given the above arguments for the WAM, it might seems obvious to adopt the WAM as the compilation model without further questioning. However, this is not so when the characteristics of current relational DBMSs are brought into consideration. There is a fundamental difference in the principles of operation of the WAM and the query evaluation engine of relational DBMSs. Because of this and prior to the construction of a new system based on an integration of the technologies discussed, a close examination of their possible interaction is relevant.

2.1. Warren Abstract Machine - WAM

The WAM [22] was conceived as a machine to be implemented in software or hardware. It is geared towards term compilation as opposed to goal compilation. The assumption is that given a goal to evaluate, a failure can be detected at an early stage in the computation so avoiding a lot of unnecessary work.

Thus, during the process of compilation, one WAM instruction is generated for each Prolog term. For example, the clause:

\[ p(a, b). \]

is compiled into:

\[
\begin{align*}
&\text{Get Constant, Atom}_a, 0, \\
&\text{Get Constant, Atom}_b, 1,
\end{align*}
\]

where the first Get Constant instruction deals with atom -a and the second Get Constant is the code generated for the atom - b.

In addition to the instructions generated for each term, some control instructions are added for procedure calls, backtracking, etc. For example, for the procedure:

\[
p(X, Y) :- \ldots, q([X] \ldots), \ldots, r(p(a, b)).
\]

the WAM code produced would be:

\[
\begin{align*}
&\text{Try Me Else} \ldots \\
&\text{Put} \ldots \\
&\text{Call q} \\
&\text{Retry Me Else} \ldots \\
&\text{Trust Me Else Fail}
\end{align*}
\]

where the instructions that control backtracking (Try Me Else, Retry Me Else and Trust Me Else Fail) surround the code for the clauses of procedure p, e.g. Put . . . Transfer of control to procedure q is implemented by the instruction Call.

The WAM is a tagged architecture designed for the support of a very flexible type system. Thus, a compiler for Prolog when doing unification will not only be checking values, but also the type of the data. For instance, the instruction Get Constant checks that the atomic type and the value can indeed be unified.

2.2. Relational Engines

In contrast to the WAM model, relational engines use a goal based strategy to answer queries. The basic premise here is that typical data base computations are bound by the transfer of data from secondary to main memory (and vice versa). It is also assumed that the time needed to read (write) a portion of a block of data from/to disc is the same as to read (write) the whole block. This is true of current disc technology, however small that portion of a block might be. Because of this, it makes sense to process a query in a goal oriented manner often consuming more cpu time but reducing the amount of data traffic to/from disc.

In addition, relational systems support a more restricted set of data types. Only atomic types are supported and they apply to attributes rather than individual terms. This considerably simplifies the handling of types, since type information does not need to be carried alongside individual atomic terms. To store the type information together with the value in a relational DBMS would represent a significant and unnecessary extra cost in time and space, since no support for general unification is required in a conventional relational DBMS. Hence, relational DBMSs implement type "checking" by means of a separate catalog, e.g. the relation and the attribute relation in Ingres [17] which at run time is used to interpret the data values brought from disc.

2.3. Interaction

In a relational engine, to process the answer for the query \( ?- p(a, X) \) using only facts for the procedure \( p/2 \), one would need a procedural program\(^1\) of the form:

\[
\begin{align*}
&\text{open rel} (\text{Descr}, "p") ; \\
&\text{set key} (\text{Descr}, \text{Query params}) ; \\
&\text{for} (\text{first tuple}(\text{Descr}) ; \\
&\text{more}(\text{Descr}) ; \\
&\text{next}(\text{Descr})) \\
&\text{get tuple}(\text{Descr}, \text{Tuple}) ; \\
&\text{further process Tuple */} \\
&\text{call inference engine */} \\
&\text{unify( Descr, Tuple) ;} \\
&\text{close rel}(\text{Descr}) ;
\end{align*}
\]

Firstly, a descriptor for the relation \( p \) is set up by opening the file for \( p \) and collecting all the information about types for the attributes. The summary information in the descriptor Descr is completed by adding the search parameters to it. The procedure first tuple(Descr) sets the descriptor pointing to the first qualifying tuple, and from there on it loops retrieving one tuple at a time, checking whether it qualifies and or not and unifying, if necessary. Following the processing of the last qualifying tuple, the loop is abandoned and the file for the relation \( p \) is closed. The program illustrates the way in which the relation \( p \) is seen as a sequential file (the use of indices on the appropriate attributes achieve this). Assuming the use of a buffering mechanism in the procedure get tuple(.), a region of the file is read into memory in blocks of several tuples which in turn get processed one at a time. In this manner, in the time it takes to read a block of data containing several tuples, the previous block can be processed. Notice that the type information has been set up in the descriptor Descr only once, before entering the loop. Also notice that unification is a one sided process, i.e. an interpretation of the array of bytes in Tuple.

\(^{1}\) The C programming language is used to illustrate algorithms at this low level.
The process illustrated by the program above is unsuitable for the processing of structured data types of a general nature. Relational DBMSs support only atomic data values, e.g., integer, real, string of characters of fixed length, etc. Some adjustments could be made to accommodate some structured data types, but the mechanism would remain as unsuited for the general case. In particular, the storage and retrieval of clauses and procedures does not fit, not to mention lists and other structured data types. Because of this, we were forced in Educe to represent general Prolog terms in the EDB in their source form as fixed length strings of characters. Although some compression can be used, source representation is wasteful of space. More seriously, it imposes an arbitrary limit to the length (in number of usable characters) of a Prolog term and it affects performance because of the parsing required prior to the use of a term.

In summary, high performance engines for relational systems and for logic deductive systems are labile, by their very nature, to conflict with each other if put together without careful analysis of their respective purpose and function. From the above discussion it is also clear that a suitable strategy 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. This strategy is confirmed by our tests which show that the time split between I/O and cpu usage [7] is very different to the one in relational systems. In our case, often the computation of answers to queries over flat relations, becomes cpu bound - even more so, in the case of rules.

3. The Fundamentals of an Architecture

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.

3.1. Management of Compiled Code in the EDB

Our tests in Educe and KB-2 showed that amongst the most important factors affecting performance in this class of systems is the parsing and assertion in main memory of clauses making up the procedures stored in the EDB. As mentioned earlier on, the use of a conventional relational DBMS to control the EDB leaves the interpreter with little choice but to store clauses in their source form as a string of characters.

Even worse, this penalty in performance per transaction is often multiplied several times over during a given period of time. Procedures maintained in the EDB have normally a transient nature (in main memory) while in use. Thus it is quite possible to have one procedure make several entry procedures several times in the course of one session.

Once the assertion of clauses has taken place, their usage represents yet one more significant factor affecting performance. Valuable information about the asserted procedure cannot be used to optimize its execution, e.g. indices as defined in [25].

In the inference engine for Educe, the performance deterioration due to parsing and assertion 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 can 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. In brief then, the component parts of the inference system are:

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 runnable in Educe’s virtual machine.

3. A very fast emulator. This emulator, derived from the WAM, interprets the code delivered by the dynamic loader.

To further quantify our previous assertions about performance, we should mention that tests in the compiler system show that about 90% of the time needed to compile a program is used by lexical analysis, parsing and memory routines, and only about 10% is used by code generation. If we equate this 10% to the time needed by the dynamic loader to resolve associative addresses (a simpler activity than code generation), we can then clearly see the potential gain to be achieved by storing compiled code in the EDB.

3.2. Flow Control

Although the program to retrieve facts as outlined in the previous section would handle queries in a very simple way, in this system it is not sufficient to handle queries in a deductive environment as offered by the programming language Prolog. One of the main reasons for this is the lack of a control mechanism to restore previous states of computation.

3.2.1. Backtracking

Backtracking is implemented in the 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 can 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, rather than to introduce additional control features for the implementation of backtracking in the relational engine. 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: how is backtracking to be 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 [19] have asserted that choice point references are the single most significant contributor to the total number of data references. In the study of Touati and Despain [19] an average of 52% of data references are identified as choice point references. Therefore, one can reasonably assume that this is a major point to consider when designing a highly efficient system.

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 centre 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 match 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 unwanted clauses and in some cases avoids the creation of choice points for these non-deterministic procedures.

363
3.2.2. Use of Indices

Since compilation to produce WAM instructions is geared towards evaluation of one term at a time, it makes good sense to index always on the first argument\(^2\), according to data type and value. This has the desirable effect of identifying failures early on. 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 comparatively poorer than those obtainable with a good index mechanism in relational systems. The reason for this is that usually 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 a relational DBMS - indexing over the type of the term, is very effective in an inferential engine. The 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\(^4\) 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.

3.2.3. Unification

The storage of compound terms and in particular of rules in source form in the EDB not only represents a penalty in performance due to parsing, and code to support indexing in main memory is added by the dynamic loader, together with other control code.

Since compilation to produce WAM instructions is geared towards evaluation of one term at a time, it makes good sense to index always on the first argument\(^2\), according to data type and value. This has the desirable effect of identifying failures early on. 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 comparatively poorer than those obtainable with a good index mechanism in relational systems. The reason for this is that usually 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 a relational DBMS - indexing over the type of the term, is very effective in an inferential engine. The 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\(^4\) 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.

In Educe\(^4\) 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.

To eliminate this form of performance penalty, Educe\(^4\) 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. This is further improved with specific machinery to support a strongly typed sub-language\(^9\). A more detailed account of the implementation of pre-unification is given in the next section.

3.3. Memory Management

The extent and complexity of the problems in this area preclude us from an in-depth discussion of them here. However, we shall comment on two of the most important aspects, and refer the reader to\([8]\) and\([9]\) for the rest.

3.3.1. Dictionary

In implementations of the WAM, the dictionary is a hashed table which keeps information about atoms and functors. The hash values are computed from the name and arity of the functor (for an atom, the arity is zero). Once a slot in the dictionary has been allocated to a functor, the index into the dictionary is used as the unique identifier for the functor. Obviously, unification on this unique identifier is going to be several orders of magnitude faster than using string comparisons, so this technique was also adopted in Educe\(^4\).

Clearly, a good design and implementation of the dictionary is essential to achieve fast compilation of programs. It is also important for overall performance of the run-time system, since many of the features of the dictionary affect the working of other parts of it, such as the remainder of the memory management subsystem and unification. A good design should consider the frequency of use of atoms and functors, their space occupancy and the speed with which is required to access, insert and delete them.

To benefit fully from unification based on the comparison of unique identifiers, we adopted the following principles to design the dictionary and its associated algorithms in Educe\(^4\):

1. The dictionary should provide unique identifiers for atoms and functors, so that the identifiers alone are used for unification. This is a standard technique in compilers: it seeks to obtain major gains in performance during unification and a reduction in the space needed at run time.

2. Space used by the dictionary should be minimized.

3. Space should not be wasted. Deleted items should be garbage collected and re-used.

4. Once an entry in the dictionary is made for a particular atom or functor, it should not be removed. Since the index in the dictionary is used as the unique identifier for the atom, if the atom were moved it would make the code for procedures in which it appears unusable. The cost in time of re-labeling atoms in the code for procedures is too high as to be done on the fly.

5. The dictionary should be extensible. In the inference engine there is no practical limit to the number of active atoms and functors in a run (session). Potentially in a system with facts and rules in the EDB, millions of atoms could be activated while processing one single query.

6. Searches of atoms and functors in the dictionary are always made for an exact match. Hence the data structure to be used should optimize equality searching.

7. For obvious reasons Key-To-Address transformation should be as fast as possible.

8. Long chains of pointers should be avoided. They slow searching, insertions and deletions. In addition, they increase the space required per entry.

Unfortunately, these principles easily lead to the use of counteracting techniques. The first contradiction - the use of a position index as a unique identifier (point 4) and garbage collection (point 3), is easily solved by re-using the slots containing deleted atoms, without attempting any relocation of active atoms. The second contradiction is more serious: it arises from our requirement for an extensible dictionary (point 5). The best

---

\(^2\)Attribute in relational terminology

---

3.3. Memory Management

The extent and complexity of the problems in this area preclude us from an in-depth discussion of them here. However, we shall comment on two of the most important aspects, and refer the reader to\([8]\) and\([9]\) for the rest.

3.3.1. Dictionary

In implementations of the WAM, the dictionary is a hashed table which keeps information about atoms and functors. The hash values are computed from the name and arity of the functor (for an atom, the arity is zero). Once a slot in the dictionary has been allocated to a functor, the index into the dictionary is used as the unique identifier for the functor. Obviously, unification on this unique identifier is going to be several orders of magnitude faster than using string comparisons, so this technique was also adopted in Educe\(^4\).

Clearly, a good design and implementation of the dictionary is essential to achieve fast compilation of programs. It is also important for overall performance of the run-time system, since many of the features of the dictionary affect the working of other parts of it, such as the remainder of the memory management subsystem and unification. A good design should consider the frequency of use of atoms and functors, their space occupancy and the speed with which is required to access, insert and delete them.

To benefit fully from unification based on the comparison of unique identifiers, we adopted the following principles to design the dictionary and its associated algorithms in Educe\(^4\):

1. The dictionary should provide unique identifiers for atoms and functors, so that the identifiers alone are used for unification. This is a standard technique in compilers: it seeks to obtain major gains in performance during unification and a reduction in the space needed at run time.

2. Space used by the dictionary should be minimized.

3. Space should not be wasted. Deleted items should be garbage collected and re-used.

4. Once an entry in the dictionary is made for a particular atom or functor, it should not be removed. Since the index in the dictionary is used as the unique identifier for the atom, if the atom were moved it would make the code for procedures in which it appears unusable. The cost in time of re-labeling atoms in the code for procedures is too high as to be done on the fly.

5. The dictionary should be extensible. In the inference engine there is no practical limit to the number of active atoms and functors in a run (session). Potentially in a system with facts and rules in the EDB, millions of atoms could be activated while processing one single query.

6. Searches of atoms and functors in the dictionary are always made for an exact match. Hence the data structure to be used should optimize equality searching.

7. For obvious reasons Key-To-Address transformation should be as fast as possible.

8. Long chains of pointers should be avoided. They slow searching, insertions and deletions. In addition, they increase the space required per entry.

Unfortunately, these principles easily lead to the use of counteracting techniques. The first contradiction - the use of a position index as a unique identifier (point 4) and garbage collection (point 3), is easily solved by re-using the slots containing deleted atoms, without attempting any relocation of active atoms. The second contradiction is more serious: it arises from our requirement for an extensible dictionary (point 5). The best
strategy in time and space for searching on equality is a hashed table, except in the case where all the possible keys are known in advance. Since we do not know this, we adopt the hashed table solution for the dictionary. Our requirement for an extendible table seems to imply that we should adopt an open hash type of table. Unfortunately, this would fail the requirement of uniqueness for the index (point 4), since several atoms (functions) could hash to the same index (bucket) in the dictionary. This major contradiction in the requirements is brought about by the use of compiled code in the EDB. If there were a way of doing garbage collection fast enough to handle these facts and rules kept in secondary storage, the problem would not arise.

The second of the contradictions leaves us with a situation where, on the one hand our requirement for an extendible table indicates that open hash should be used, but on the other hand our requirement for uniqueness of addresses points toward a closed hash table as a solution. Our solution combines the two approaches by adopting a segmented table to which segments may be added on demand, but each segment is by itself a closed hash table.

This closed segments hash table provides unique identifiers by a concatenation of the segment number and the index (generated by the hash function). Initially a suitable sized segment is allocated for use by the dictionary. As this segment passes a high water mark for occupancy, e.g. 70%, a new segment is allocated and chained to the previous one. This process is repeated each time all the existing segments pass the high water mark. The segment with lowest occupancy is designated as the hot segment and new insertions are only made in this segment. This is an attempt to gradually balance the occupancy amongst segments, thus minimizing the average length of collision chains within segments. A hash table entry with an occupancy rate below the high water mark. This technique effectively garbage collects segments.

3.3.2. Garbage Collection

Continuous operation over long periods of time required by data and knowledge base systems, demands full garbage collection. In Educe*, only main memory need to be garbage collected, garbage collection of the secondary store is avoided by allowing only the use of associative addresses in the compiled code kept in internal memory and by the use of dynamic storage techniques - the Bang file system [13, 14], for storing the items.

The requirement of continuous operation over long periods of time, although desirable for logic languages, is not considered compulsory by designers and implementors of them. In fact only minimal garbage collection, if any, is implemented in interpreters and compilers for these languages. In particular in implementations based on the WAM, although the WAM itself would do some garbage collection of stacks, other areas would be totally neglected. For instance, the assumption is made that entries deleted in the dictionary do not take much space and hence it is not worth the effort to design a mechanism for recovery of that space. This may be true in a Prolog system running in main memory with no connections to the outside world, but certainly not true in the context of an inference system with an EDB.

Because of the potential for very large programs that Educe* introduces and the likely long delays caused by complete garbage collection, we decided to spread the delays along the normal processing of queries. Thus, garbage collection is done in an incremental manner with facilities to temporarily disable it in those cases where severe time constraints apply, e.g. critical regions of real time applications.

In general, garbage collection activity is triggered by the activity of the stacks, compilation of programs and assertions/deletions of externally maintained code. Space is recovered in the stacks by its own discipline, in synchrony with calls to and exits from procedures. Tail recursion optimization takes care of space re-use in the appropriate cases, so allowing for infinity loops. The global stack which is used to dynamically build complex data structures, is garbage collected by means of a sliding incremental garbage collector. The garbage collection of dictionary and general heap (main memory used for storing strings of characters making up the name of atoms or functions) is triggered by direct insertion at goal invocation and by erasure after they have been executed (allowing for delays due to semantics). The heaps maintain a free list of blocks of memory for re-use.

More details about garbage collection, in particular garbage collection of the stacks, are given in [6].

4. Implementation of Pre-unification

In order to give a more comprehensive understanding of the working of the Educe* system, we present in this section an overview of the implementation of pre-unification on external storage.

In addition to the structures required to manage internal memory, two general tables are kept by Bang: the external dictionary and the clauses relation. Also, each procedure kept in external storage is marked as such in the internal procedures table and a specific relation in Bang is used to gain access to the clauses making up the procedures. A description of these structures follows:

1. Procedures Table. All procedures whether internal or external have an entry in this table. External procedures are marked as such.

2. External Dictionary. This is a table managed by Bang to keep information about atoms and functors in external storage. An entry here has the string of characters making the name of an atom or functor, its arity and a computed hash value. The hash value is computed by applying the hash function of the internal dictionary, without clash resolution, to the atom (functor) concerned. The computed hash value is used to assist pre-unification in Bang and the strings of characters are used in range queries.

3. Procedures Relation. For each procedure kept in external storage one Bang relation is set up. This relation has one tuple for each clause in the procedure. For each argument in the head of the procedure there is one attribute. In addition, there is the clause_id and the code attributes. The clause_id attribute uniquely identifies the clause within the procedure and the code attribute is a boolean value indicating whether compiled code is associated with the clause. Attributes can have as valid format: integer, real, atom, list, structure and tagged integers and reals occupy only the space required to store the actual value of the integer/real. Atom type attributes store the hash value used in the external dictionary. Tagged attributes store a tagged integer value. Lists, structures and clauses with a body make use of the clauses relation to keep the compiled code associated with them.

4. Clauses Relation. This is also maintained as a Bang relation. It has three attributes: procedure_id, clause_id and relative_code. This last attribute contains compiled code with relative addresses, i.e. references in it, are to the external dictionary instead of the internal dictionary. The other two attributes are used to uniquely attach the code to one clause in a procedure.
The above scheme of operation is instrumented by execution from inside Bang of the unification routines in the deductive engine of Educe*. Bang can directly execute compiled code kept in the clauses relation. However, it should be noticed that the code kept in secondary memory only has associative addresses, and hence, successful execution of this code does not necessarily mean that the clause would satisfy the user's goal. Successful execution is a necessary but not sufficient requirement. Similarly, in the presence of deeply nested structured terms, it is possible to select a clause by executing only the code corresponding to the highest levels of nesting. At the time of writing, we have not yet established a definitive strategy for deciding how much of the code should be successfully executed, before a clause is selected for refined processing. This we believe is a matter for empirical experimentation, still to be done.

It should be noted that the scheme of operation described above covers the special case of ordinary relations as found in conventional relational DBMS. In this case, the attribute code of the procedures relation is set to false and only atomic formats are allowed for the attributes. This allows for the processing of such relations by means of conventional relational operations, if so required by the programmer. For this, see the relational operators of Educe* in [9].

Our scheme of operation gives the flexibility to switch between a goal-oriented and a term-oriented evaluation strategy, according to circumstances. Even better, programmers can mix the two without performance penalties.

5. Preliminary Evaluation of Performance

To the best of our knowledge there is as yet no other system offering the same or similar functionality to the one afforded by Educe*. It is not therefore possible to establish the relative performance of our knowledge base a supported by Educe* that have no homologues in other system, e.g. the execution of compiled logic programs kept in secondary memory. Even direct comparisons with systems offering some of Educe*'s functionality, such as relational DBMSs and/or stand alone Prolog compilers, may be misleading. For example, to compare retrieval of a single tuple in Educe* against the same operation in a conventional relational DBMS is unfair to the relational system. The whole design of a relational system is geared towards the evaluation of complete sets. Similarly, examples to the contrary can also be given. Despite all of this, we still wanted to have some performance targets. Thus, we set them for Educe* at a level to be competitive with the best of relational and logic programming systems on equivalent operations.

We have run some tests on the basic machinery of Educe*. The results are very encouraging and tend to confirm our basic reasoning. The tests cover three areas: Educe* as a step forward from systems based on the integration of components in the Educe fashion. Educe* used as a conventional relational DBMS, and Educe* used as a Prolog compiler. For this, three batteries of tests were used. The first one is the MVV application with which we tested earlier on our Educe system [7]. The second one is based on the Wisconsin benchmark for relational data bases [3], and the third one is the database integrity check task, an application developed by F. Bry and tested by M. Dahmen, both at ECRC. We emphasize that we do not consider these tests as definitive measures of the performance of Educe*, in any way. However, we feel that they provide us with important validation data and some reasonable feedback to continue our work on the development of Educe*, in particular on performance improvements.

The configuration used for the tests was a Sun 3/280S (25 Mhz) computer with a Hitachi disc with 850 Mbytes, running the Unix - Berkeley 4.0 version of the operating system. For the kernel of Educe*, the total stacks space allocated was 2 Mbytes and a dictionary of 32000 entries per segment. This configuration creates a process of about 2.5 Mbytes which runs comfortably in our small Sun workstations (disks with 4 Mbytes). In the case of a file server Sun computer no noticeable deterioration in throughput is caused by several concurrent users (>7) with the above configuration. Of course, the relatively limited amount of real memory allocated to the process often causes activation of the garbage collector. This activity is accounted for in the figures given below. Also, in some of the tests a Sun 3/60 diskless workstation was used.

5.1. Muenchner Verkehrs Verbund

The Muenchner Verkehrs Verbund (MVV) knowledge base is a realistic example based on the public transport system of the city of Munich. The MVV combines the use of buses, underground trains, commuter trains and trams into one transport network. This example highlights the requirements for KBMSs. Our tests are a set of queries on how to get from one part of the city to another, starting at a given time. The problem is described in more details in [7].

We ran two classes of queries:

1. Class 1. Simple queries: involving travel between adjacent major nodes with minimal choice of means of transport involved.
2. Class 2. Involved queries: to find travel routes between major nodes, restricted to not more than one change and with many means of transport to choose between.

The experiments were run with the rules for the program held in internal storage and the facts about type of transport, places and timetables held in three relations on external storage:

1. Relation location2: with arity 2 and 2307 tuples.
2. Relation schedule3: with arity 11 and 8776 tuples.
3. Relation schedule2: with arity 5 and 7260 tuples.

A sample of ten queries from each class was run on the application and the average time achieved was taken. The times are in seconds spent by the process in obtaining the answer.

Educe* - MVV times

<table>
<thead>
<tr>
<th>Query</th>
<th>First Run</th>
<th>Second Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td></td>
<td>(3)</td>
<td>(4)</td>
</tr>
<tr>
<td></td>
<td>(5)</td>
<td>(6)</td>
</tr>
<tr>
<td></td>
<td>(7)</td>
<td>(8)</td>
</tr>
<tr>
<td></td>
<td>(9)</td>
<td>(10)</td>
</tr>
</tbody>
</table>

We ran the tests for a second time, so that possible distortions in timings due to initial loading of bufferings could be spotted. From the figures, there is no evidence of significant distortions. We split the time into cpu time and system time (as indication of I/O activity) to try to assess the relative importance of I/O and cpu usage. In general, we found the impact of I/O very low in this application. To gain more evidence of this, we also ran the application on a single user configuration. We found no significant differences between cpu plus system time and clock elapsed time.

---

*We also include the time for Educe. It should be noticed that the algorithms used are not the same. However, the classes of queries in which large differences could be due to this factor, are excluded from this report. The new algorithms for the MVV knowledge base have been developed by A. Tomaso, at ECRC
5.2. Wisconsin Benchmarks
In order to have an indication of Educe's relational capabilities, we selected some relevant tests from the Wisconsin benchmark. The tests run were:

1. Selection with 1 percent selectivity over a 10000 tuples relation.
2. Selection with 10 percent selectivity over a 10000 tuples relation.
3. Select 1 tuple to screen from 10000 tuples relation.
4. Two-way join of two 10000 tuples with selection over one of them.
5. Three-way join of two relations with 10000 tuples and one with 1000 tuples. In addition, selections were done over the two 10000 tuples relations.

Each one of the tests was run several times and each time the query was expressed in a different format. The results of running the tests were:

**Educe* - Wisconsin**

<table>
<thead>
<tr>
<th>Query</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocess (in milliseconds)</td>
<td>3482</td>
<td>431</td>
<td>258</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>17716</td>
<td>3192</td>
<td>2175</td>
<td></td>
</tr>
<tr>
<td>Buffer read/write</td>
<td>52</td>
<td>31</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Total I/O activity</td>
<td>48242</td>
<td>2456</td>
<td>1853</td>
<td></td>
</tr>
<tr>
<td>Average Times (secs)</td>
<td>83713</td>
<td>16824</td>
<td>15470</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2a - times (seconds)**

<table>
<thead>
<tr>
<th>Query</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocess</td>
<td>10</td>
<td>4.2</td>
<td>0.08</td>
<td>0.50</td>
<td>2.13</td>
<td>0.50</td>
<td>2.63</td>
</tr>
<tr>
<td>CPU</td>
<td>10</td>
<td>0.02</td>
<td>0.1</td>
<td>0.03</td>
<td>33.16</td>
<td>2.75</td>
<td>33.91</td>
</tr>
<tr>
<td>Buffer read/write</td>
<td>4</td>
<td>27.03</td>
<td>7.07</td>
<td>34.10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 2b - I/O freq.**

For each query class n different variants of it were run. Also the total number of buffers accessed and total numbers of read and write pages per class of query were recorded.

5.3. Database Integrity Checking
The Integrity Checking (IC) program is an example of database integrity checking. It consists of a small database and a relatively short Prolog program. The database of facts is stored in interpreted form in Prolog with the exception of a single relation with four thousand tuples which is given special treatment. The database contains:

1. One relation with about 4000 tuples, each with seven fields.
2. Fifteen relations with up to 20 tuples, with one or two fields.
3. One relation with about 50 tuples, each with two fields.
4. Seven rules.
5. Five integrity constraints of very different complexity.

The test is divided in three parts: full test is a naive approach to integrity checking, with all constraints being checked against the changed database; preprocess computes a specialisation of the integrity constraints, and it does not require any access to the facts of the data base; and partial test uses the specialisation to check the integrity of the updates.

The times given below are only for preprocess. This part of the test isolates the more conventional use of a Prolog compiler.

**Integrity Constraints Checking**

<table>
<thead>
<tr>
<th>Machine</th>
<th>Sun client</th>
<th>Sun server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>GC</td>
<td>E*</td>
</tr>
<tr>
<td>1</td>
<td>724</td>
<td>380</td>
</tr>
<tr>
<td>2</td>
<td>1079</td>
<td>575</td>
</tr>
<tr>
<td>3</td>
<td>2803</td>
<td>1420</td>
</tr>
<tr>
<td>4</td>
<td>3483</td>
<td>2890</td>
</tr>
<tr>
<td>5</td>
<td>4258</td>
<td>2140</td>
</tr>
</tbody>
</table>

**Table 3 - Preprocess (in milliseconds)**

GC : A Good Prolog Compiler
E* : Educe*

The times for full test and partial test are given in [12]. They are not included here since they only confirm the results of our previous tests on relational performance.

5.4. Some results
These provisional tests of the Educe* kernel tentatively confirm the validity of our reasoning in designing the basic architecture. The results of the tests show that we have considerably improved performance in the case of the MVV application. While in the case of the Wisconsin benchmarks we can easily match the performance of the relational DBMSs available at our installation, often outperforming them. The last test, the preprocess part of the IC program shows that the performance of Educe* also compares well with the performance of good Prolog compilers. We estimate that with some tuning and some minor modifications, we should be able to further improve performance by some 50-100%.

Amongst the important points to notice, the relative importance of cpu time over I/O time is very significant. A relatively large memory increases the possibilities for buffering, so making the whole system more sensitive to cpu activity.

The effect of cpu predominance was confirmed when we ran queries of class 1 and 2 on a discless workstation. The time deterioration can be partly attributed to the degradation of cpu performance, i.e. from a M68020 processor at 25 MHz (4 MIPS) to the same processor running at 20 MHz (3 MIPS). This is shown in table 3, below.

---

*This test was designed and run by M. Dahmen. We quote from his report [12].
A second point of importance is that, although not specifically mentioned in the tables above, the garbage collector was constantly invoked and considerable amounts of memory were recovered most of the time. Since garbage collector activity is accounted for in the figures given above, it can categorically be said that its effect on overall performance is negligible. Any argument for not including a garbage collector, based on the deterioration in performance that garbage collection might cause, is thus, demonstrably false. The cost paid in loss of functionality through not having garbage collection is far too high. Even when it is possible to grab massive amounts of memory at the expense of other users of the system, it is still not possible to offer some continuity of operation over a period of time without a garbage collector. In our examples, the use of only 2 MBytes for stacks and suitable amounts of memory for heaps and dictionary permits a good throughput. However, this relatively economical use of memory is only possible because of Educe*’s garbage collector.

6. Conclusions and Further Work
Our review of previous experimental work on the integration of current technologies in the fields of logic programming and relational DBMSs shows that, although they offer a most promising ground on which to build the Knowledge Base Systems of the future, their actual integration is a rather complex task. This is particularly true if the performance of KBMSs is to reach product standard as opposed to simple exploration of functionality. Systems of the Educe type essentially sought to investigate issues of functionality. With Educe* we attempt a production system with which to implement Deductive Database Systems and Knowledge Base Management Systems; systems in which the management of rules is as important, if often not more important, than the management of facts.

At the centre of the strategy for high performance in Educe* are the storage of compiled code in the EDB and the careful integration of the principles of relational and logic technologies. It is not sufficient to loosely integrate two existing systems, each based on one of the technologies. This only serves the purpose of functional exploration. For real production systems the design of an architecture from first principles is required. Precisely this was done in Educe*. "Deductive" implementation is in its final stages, and our early tests show that the functionality and performance of Educe* is comparable to the one offered by good relational DBMSs on problems suited to relational systems. The same is true in comparisons against logic languages, in particular Prolog compilers. The difference is that the technology used in Educe* is capable of handling deduction efficiently over very large sets of facts and rules.

As for further work, we continue to expand functionality with work on data types [9] and on performance improvements. Optimization of queries in hybrid systems of the Educe* type acquires a very different perspective to that adopted by Prolog compilers or relational DBMSs. Tests done on Educe* [4, 5] confirm this tendency which our early tests on Educe* confirm. The basic assumptions used to define and choose strategies in relational DBMSs and Prolog compilers need to be reconsidered when dealing with a hybrid system which integrates (amongst others) the functionality of the two former system types. We believe that Educe* is only at the beginning of the task.

Acknowledgements
I would like to thank fellow members of the Knowledge Base group at ECRC for their contributions in discussions relating to this work. In particular, I am grateful to Jean-Marie Nicolais for his support of this project, Philip Pearson and Geoffrey Macartney for their work on the compiler and their help with the tests, Peter Bailey for a very efficient implementation of the garbage collector, Michael Dahmen for the IC test and for such a powerful debugger, Pierre Coste and Xavier Savalle for their part in the Interface Bang, and last but not least to Mike Freeston for the Bang file - the most efficient file system to support a deductive capability. My thanks to all of them also for providing friendship, cheerfulness and general support.


References

[1] Bancilhon, F. and Ramakrishnan, R.
An amateur's introduction to recursive query processing strategies.

[2] Beeri, C.
Recursive query processing.

Benchmarking Database Systems - A Systematic Approach.

On the Evaluation Strategy of EDUCE.

EDUCE - A Marriage of Convenience: Prolog and a Relational DBMS.

[6] Bocca, J. and Bailey, P.
Logic Languages and Relational DBMSs - The point of convergence.

On Prolog-DBMS Connections: A Step Forward from EDUCE.

KB-PROLOG, A Prolog for Very Large Knowledge Bases.

[9] Bocca, J.
EDUCE - A logic programming system for implementing KBMS's.

A Portable Prolog Compiler.

Query Evaluation in Recursive Databases: Bottom-up and Top-down Reconciled.

[12] Dahmen, M.
Performance Evaluation of Different Prolog systems for a database integrity checking task.

[13] Freeston, M.
The BANG File: A New Kind of Grid File.

[14] Freeston, M.
Grid files for efficient Prolog clause access.


Extended Prolog for Order-Sorted Resolution.

[17] Stonebraker, M., Wong, E., Kreps, P. and Held, H.
The Design and Implementation of Ingres.

Memory Performance of Prolog Architectures.

[19] Touati, H. and Despain, A.

[20] Vieille, L.
Recursive query processing: The power of logic.

[21] Wallace, M.
KB2: a Knowledge Based System embedded in Prolog.

An Abstract Prolog Instruction Set.

369