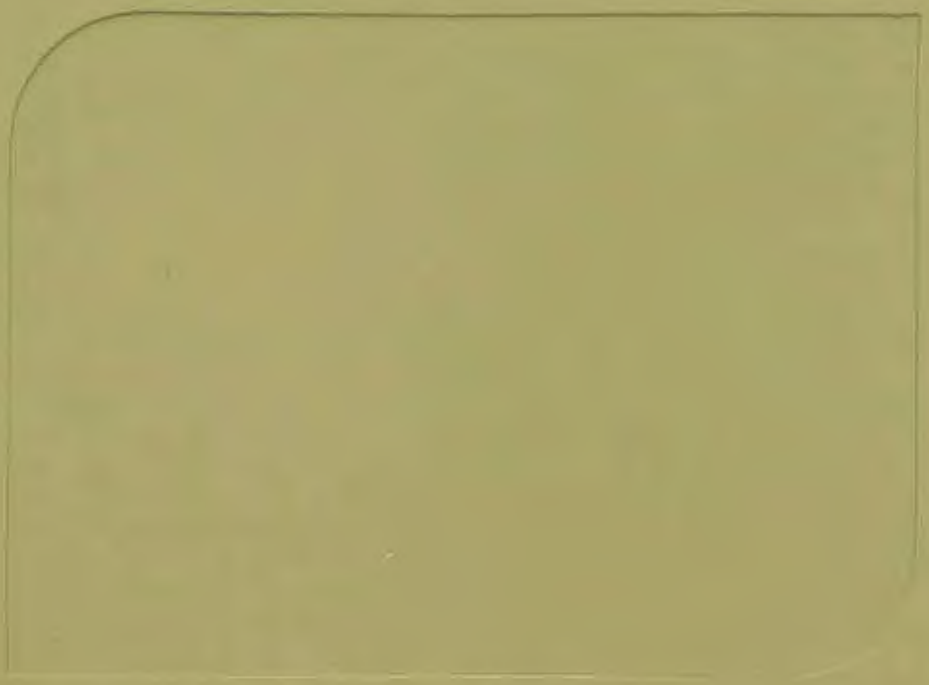


# BIM-PROLOG

Joint Project between
BIM
and
Department of Computer Science
Katholieke Universiteit LEUVEN

Sponsored by DPWB/SPPS
under grant nr KBAR/SOFT/1

# The Interaction between Prolog and Relational Databases

by

Raf VENKEN *

Internal Report
BIM-prolog IR6

September 1984

* BIM
Kwikstraat 4
B-3078 Everberg Belgium
tel. +32 2 759 59 25

** Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A
B-3030 Heverlee Belgium
tel. +32 16 20 06 56

DPWB = Diensten van de eerste minister : Programmatie van het Wetenschapsbeleid.
SPPS = Services du premier ministre : Programmation de la Politique Scientifique.

# THE INTERACTION BETWEEN PROLOG AND RELATIONAL DATABASES.

by

Raf Venken

B.I.M.
Kwikstraat 4
B-3078 Everberg
Belgium

## 1. Abstract.

This paper summarizes the results of the study of the different compilation techniques to transform Prolog queries in conjunctions of data base calls. A brief survey of existing compilation techniques is given, followed by a short intro uction to partial evaluation. Then we demonstrate how this technique could be used as an alternative to the previous compilation techniques.

We further investigated the different ways and techniques to effectively integrate an existing Prolog and data base system. We give a survey of the different concepts and state the requirements for data bases to allow such integration. The impact on the Prolog system is analysed.

## 2. Introduction.

Most applications written in Prolog use only a limited number of rules, which can therefore be kept in the internal data structures of a Prolog system. However, depending on the nature of the application, there can be a considerable amount of el mentary facts that cannot be mastered in core and thus would have to be kept on secondary storage.

One approach to solve this problem is to integrate a suitable file system into the Prolog interpreter. The advantage of this approach is that the access to elementary facts can be realized very efficiently by implementing a dedicated retrieval scheme for the Prolog partial match queries. The manipulation of the facts on secondary storage is invisible to the user, the access and manipulation of elementary facts is translated automatically by the dedicated Prolog system in a series of elementary manipulations of the underlying dedicated file system. Examples of this approach are described in <Ven81>, where the K.U.L.-Prolog is integrated with a multi-level B+ tree mechanism, and in <Llo83> and <Ram83> who incorporated a multi-key hashing method for dynamic file access into MU-Prolog.

The disadvantage of this approach however is that the file system is dedicated to the Prolog type of data access and is not easily accessable from existing software tools. On the other hand, the only way to access the data residing in existing and eventually huge databases is to convert them (eventually partially) into the dedicated filesystem, which can be very impractical and time consuming.

The alternative solution, which we investigate in this paper, is to use a conventional relational data base system which solves the fact storage and retrieval problem, and integrate it, as a "back-end", with the deductive part (the inference mechanism of Prolog).

## 3. Transforming Prolog queries into data base calls.

There are essentially two ways to transform a Prolog query into a conjunction of database calls. The first is the so called "compiled approach" (see work of Reiter, Kellogs et al. and Chang in <Gal78>), the second the "interpretive approach" (see <Ven81> or <Cha82>). In this section we describe also the partial evaluation technique and show how it can be used as a compromise between the former two methods.

### 3.1 The compiled approach.

In the compiled approach it is assumed that the procedural statements are non-recursive. Therefore one can apply procedural statements until all goal statements consis of relations that are known to be stored in the database. The result is then a conjunction of database calls which is passed to the database for answering. There are two methods to realize this approach, one requires some modifications to be made to the Prolog system, which essentially consists of delaying database calls until one has one conjunction of database calls that can be handled by the relational database. The second method avoids to make this changes to the Prolog system by the construction of a Prolog metasystem, a Prolog system written in Prolog, that simulates the behaviour of the former system.

The advantage of the compiled approach is that the calls to the database are clustered together in a conjunction which can be the subject of some less or more elaborated optimisation process (see e.g. <War81>). One great disadvantage of the compiled approach is the restriction to non-recursive rules. This problem has been partially solved in <Hen84> where it is shown how recursive queries can be compiled towards iterative programs. Further investigation should point out how this compilation technique could be used in a compilation of Prolog to an iterative language.

### 3.2 The interpretive approach.

In the interpretive approach one interleaves searches of the database with deductive steps. Each time a database call is encountered by the Prolog system, this call is sent to the database for evaluation, the Prolog system resumes execution with values obtained from the first resulting tuple, the remaining tuples of the solution of the query are consumed one by one on backtracking.

One possibility to implement this scheme is to change the database system in this way that it always gives the first tuple which answers the query to the Prolog system. The next tuples are provided on demand of the backtracking mechanism of the Prolog system. This approach can be implemented with a stack, all tuples are stored on a stack and consumed one by one by the Prolog system on backtracking. It appears that one only needs one stack for storing the resulting tuples of

consecutive database calls. More details concerning this approach can be found in <Ven81>.

A variant of this approach is the socalled set-oriented approach <Cha82>. In contrast with the previous approach, which typically generates the search tree one node at a time, the set-oriented approach manages the entire unification set at each node. The Prolog system has a set of substitutions instead of a single substitution at each node and the operations are performed on the whole set. The task of generating, including intersection and union of unification sets is delegated to the data base system which can handle it in an efficient way. This approach requires a considerable change in the Prolog system and a special data base system which can take advantage of some parallel processing.

Of all these approaches the stack-oriented interpretive approach seems to be the closest to the logic programming philosphy. However, in the context of an interactive natural language system for querying data bases it seems appropriate to consider an alternative method which, as the compiled approach, could permit an optimisation process on the order of data base calls. The partial evaluation technique adapted for data base manipulation seems an ideal alternative.

### 3.3 Partial evaluation.

It is very likely that in the near future a large part of the programming will be done in higher level languages. These languages seem appropriate tools for efficient problem solving but pose one serious problem : because of their high level of abstraction they seem not too appropriate to be executed efficiently on conventional sequential machines. This forces the programmer for the moment to fall back on low level programming languages or styles at the expense of clarity and programming methodology.

In <Kom81> partial evaluation of Prolog programs is investigated as a part of a theory of interactive, incremental psogramming, with the goal to provide formally correct, interactive programming tools for program transformation. These program transformation tools will play an eminent role in program optimisation.

The initial goal of partial evaluation is to transform Prolog programs into more efficient ones. T- is optimisation is accomplished by mainly three techniques : instantiate the parameters of a program by propagating values for top-level arguments through the program (perform the unification process at compile time), reduce the number of logical inferences by opening calls and by evaluating builtin predicates (builtins for short) whenever possible. Partial evaluation can be seen as a compile-time application of the basic mechanisms, which are normally applied at run-time. A Prolog program is converted to a semantically equivalent Prolog program, where unification and evaluation is already partially done, thus needing less logical inferences at run-time. In <Ven84> it is described how a partial evaluation system can be built on the basis of a Prolog meta-interpreter.

The effect of the partial evaluation on a program interacting with a

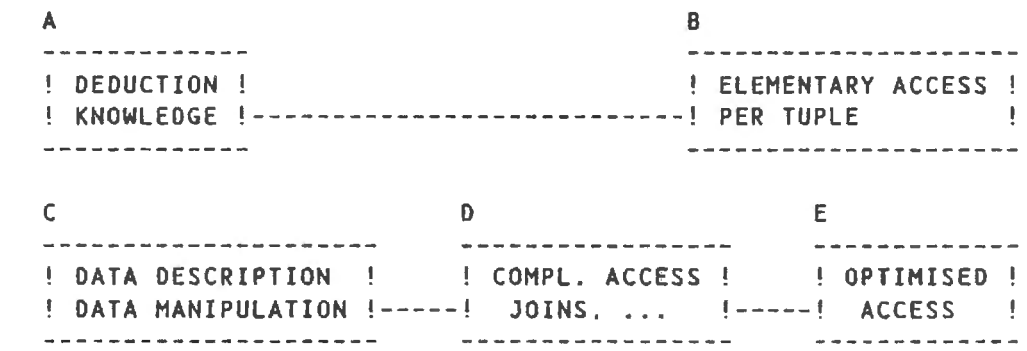
relational database, is very similar to that of the compiled approach : the transformed program consists of a conjunction of builtins (which were not evaluable at compile time), calls to recursive rules and database calls. Each recursive rule is partially evaluated to a similar conjunction of calls. As in the compiled approach the calls to the data base are clustered together and can be optimised using statistical (<War81>) or semantic information (<Kin81> or <Ham80>).

The evaluation of the resulting programs however is very similar to the interpretive approach. A normal Prolog system can be used to solve the builtins and recursive rules, the conjunctions of data base calls are transferred to the data base system, the results, a set of tuples, is again stored on a stack and consumed one by one on backtracking of the Prolog system. The change to be made to the Prolog system is exactly the same as in the interpretive approach.

## 4. Integration of a Prolog system and a relational Database system.

### 4.1 Some concepts.

In <Gal83>, the most essential conceptual blocks to build a Prolog system on one hand and a database system on the other hand are identified :




A conventional Prolog system consists of building blocks A and B, a con entional database system of the blocks C, D and E. These building blocks however can be combined in different ways for special purposes. A Prolog+ system is defined as a combination of blocks A, B and E, i.e. a Prolog system extended with a dedicated file access, which permits efficient access to a great amount of elementary facts stored on secondary devices. Prolog-DB stands for a combination of A, D and E, which permits some optimisation to be done on conjunctions of database calls. Adding building block C gives us a so called logic database wich incorporates typical database functionalities as integrity constraints, views, etc. These 3 kinds of systems have all the Prolog system as kernel. One could also start from an existing conventional database system and add a deduction component (A and C, D and E) to obtain a deductive database, and add functionalities like incomplete information and deduction rules.

### 4.2 Level of access.

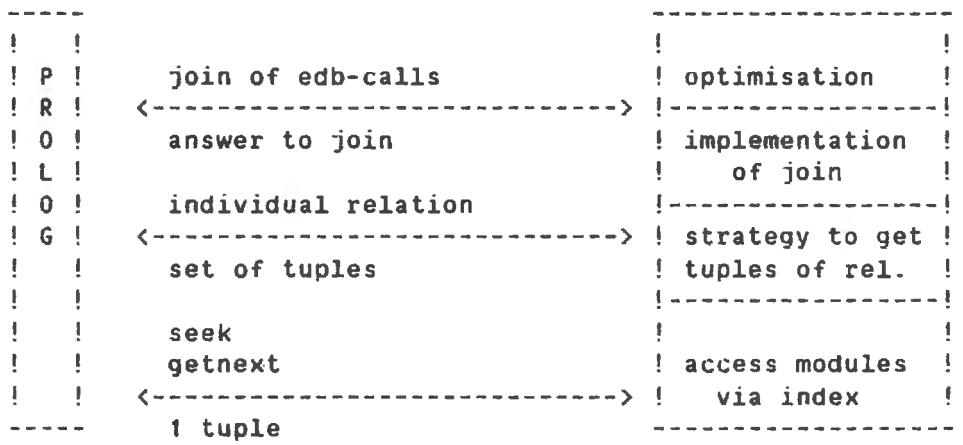
On the functional level there are different ways to access a

relational database system :

On the upper layer we could think of a system that translates Prolog database queries into a SQL-type of querylanguage. These queries could be answered by a standard database system, this would ask only a very simple interface that fits in the scheme of the compiled approach.

In the compiled approach one could also think of a lower layer of access which solves a conjunction of database queries. The standard optimisation of the database system can then be replaced by a more domain dependent or application ependent optimisation scheme residing in the Prolog system.

Even lower levels of access can be useful in the context of the development of information systems in Prolog. Primitives to query single relations, or even to manipulate filepointers and individual tuples should be available then. In this case the responsability for the optimisation process is shifted towards the application programmer. It is feasible to implement the higher levels of access in terms of this last one.




The ways to effectively integrate an existing Prolog system and a commercial database system evolve in a natural way from this scheme.

### 4.3 Prolog and SQL.

Since a database query as stated in Prolog, is essentially a conjunction of calls to elementary relations, it is evident that any relational query language can express those queries. A simple way thus to couple a Prolog system and a relational database is to use the compiled approach or the partial evaluation approach and translate the resulting conjunctions of database queries into the available database query language. On execution of the program the query is translated into the query language and transmitted to the database system, the answers on the other hand have to be translated into the internal Prolog format.

We studied this approach for the SQL/SEQUEL type of languages. The results of this investigation can be summarised as follows :

- The form of the conjunctions of database calls as they appear in Prolog correspond with the following SQL-type of query :

  SELECT <variables> FROM <union of relations>
  WHERE <set of equalities>

  This type of query is general enough to express all queries than can be expressed in the framework of Prolog. The transformation to be made is rather simple and straigthforward.

- However in this case one does not use the full power of the SQL-like language, these generally provide special constructs for special types of queries, but when these special queries are expressed in the unique Prolog form, it is not always evident or simple to make the conversion to the appropriate SQL-query.

- When using this kind of interaction, it is not possible to control the process of optimisation, in general it is not possible to switch off the optimisation process, or even to recognize (on reading the reference manual) if there is an optimisation at all. In particular it is not clear if the order of relations in the union or conditions in the set of equalities has an impact on the response time of the database system.

- The answer of the database system to the query is usually very human oriented and not machine or Prolog oriented, this implies that a transformation has to be applied on the resulting tables to convert them to the internal Prolog format.

- The only requirement imposed on the database system is that it can be invoked from a procedural language and that results of a query can be collected within that language. This technique has been tested with the integration of Ingres which offers the possibility to embed QUEL statements between the language statements) and Prolog.

### 4.4 Prolog and a conjunction of database calls.

An other way to realize an integration between Prolog and relational database systems is offered by the partial evaluation variant of the compiled approach. The Prolog program as stated by the user is converted in a number of rules each consisting of a conjunction of calls to builtins, calls to recursive rules and conjunctions of database calls. These conjunctions of database calls can eventually be transformed into an optimised one, using different techniques of query optimisation : syntactic, statistic or semantic query optimisation (see <War81>, <Ham80> and <Kin81>).

This technique requires a slight modifi ation to be made to both the Prolog and the database system.

The modifications to be made to the Prolog system are very similar as those made in the interpretive approach : when the system during the normal evaluation of the transformed Prolog program encounters a conjunction of database calls, the system transfers it for evaluation to the database system, eventually after some optimisation, the

results of the global query are furnished to the Prolog system one by one which consumes them on backtracking. As already said before, this approach can be implemented using a stack.

Onside of the database system, it should be possible to switch off the optimisation process. This implies in general that one has access to a lower layer of the dat base system (a socalled open system ; e.g. a system that offers a standard interface to some conventional programming language as Pascal or C) or to the source code of the system (in which case one can apply any desired modification).

It is very important to consider the possibilities of some par llellism : once the database system has found the first solution to the query coming from the Prolog system, the Prolog system can continue its deductions, while the database system works in parallel to find alternative solutions which can be stored on a stack to be consumed later by the Prolog system on eventual backtracking.

We essentially see two possible approaches :

- Open system : the database system includes a conventional language interface to all needed primitive actions.
  The database system can be seen as a set of subroutines which can be included in the Prolog system. The Prolog system has to be split up in two parallel processes : one which handles the normal Prolog functions, the second which would solve the database queries coming from the first process by calling the appropriate database subroutines and giving back subsequent solutions on demand to the first process.

- Closed system : if the database system offers no such language interface, the only way of integration is offered by modification of the source of the database system.
  Again we have two processes : the first is the normal Prolog system augmented with a layer of communication and synchronisation with other processes. The second process is the database system enriched with the same layer of communication and synchronisation and a modification of the standard interface : concerning the syntax of the queries it can handle and concerning the syntax and the way it gives back solutions on demand. This approach is being tested in the integration of Prolog and Unify.

### 4.5 Primitive database actions.

Depending on the application which has to be written in Prolog, it can be necessary to provide the Prolog programmer with an access to the same primitive actions as used in the 'Open system'-approach by the Prolog system itself. E.g. if one writes the optimisation module in Prolog (which is a reasonable choice), the programmer needs access to some kind of datadictionary containing statistical or meta-information, which is stored in the same way as the raw data. Access to this kind of information has to be structured and optimised by the programmer himself, using a more primitive kind of interaction, which is similar to the approach proposed in <Rie81> (however, they propose this kind of approach in the overall interaction between a

conventional programming language and a database system).

The level of interaction needed in such approach are of the following kind : positioning and moving filepointers, extracting records from files, manipulating buffers and values, etc., which are generally offered in a conventional language interface and can be implemented as desired when having access to the source code.

## 5. Implementation of a database interface at the Prolog side.

The idea behind the actual implementation, which we are finishing for the moment, is to make the database as invisible as possible to the user and to permit the connexion of any relational database, which obeys the requirements stated above, without changing the Prolog system.

### 5.1 The view of the Prolog user.

The only impact on the way the user writes its program, is the fact that he has to declare the external database relations, eventually these declarations can be generated from a datadictionary. The programs are then fed into the partial evaluation system which semi-automatically transforms and optimizes them. The calls to relations which reside on the external da abase are transformed in the appropriate formalism.

### 5.2 The view of the Prolog system.

The Prolog system activates the database process and establishes a communication channel between its own and the database process. The conjunctions of database calls are transmitted across this channel in an appropriate syntax, the database replies with a first result, i.e. a set of values which the Prolog system assigns to the appropriate variables. The Prolog sys em continues evaluation of the program, eventually containing other database calls, on backtracking, subsequent solutions to the queries are requested from the database and consumed in the same way. Eventually, on encountering a 'cut' in the Prolog program, some solutions, still pending on the stack of the database, are not needed anymore, a special discard command is transmitted then from the Prolog system to the database. In the same way, 'insert', 'delete' and 'update' can be implemented.

At first sight the implementation is straightforward : a database call is a backtrackable builtin predicate, the conjunction is transmitted across the communication channel, by numbering the occurring variables in an appropriate way, the assignment of the values is simplified. But there is one little problem : the constants coming back from the database system are to be stored in the constant table of the Prolog system, due to the fact that the amount of constants returned from the database can be huge, some precautions have to be taken. In principle there are two solutions : an appropriate garbage collector or a temporary constant table for database constants.

#### 5.2.1 Garbage collector.

When the Prolog system has an appropriate garbage collector which

cleans the constant table too, one could solve the problem of the huge amount of constants eventually coming from the database system as follows : the constants are stored in the normal constant table, whenever this table is full, the garbage collector is activated. The principle of this kind of garbage collection is very simple : constants referenced by Prolog code, active variables and structures are marked, all unmarked constants are deleted. To optimize this process, one could permanently mark the constants referenced by the static Prolog code, i.e. the compiled ode.

The disadvantage of this approach is in the case of an interactive system : depending on the size of the different tables, the garbage colsection can be time consuming, degrading thus t e immediate respons time to the user.

5.2.2 A temporary constant table.

The constants can also be stored on a temporary constant table which can be organized as a stack, since the constants will never be referenced anymore they can savely disappear from the stack on backtracking. However, in order to not increase the computing time of the unification, some precautions have to be taken in order that constants are not duplicated in both tables or in a single table : if the constant is already in the permanent constant table it should not be duplicated in the temporary one, also in the temporary table a constant should appear only once. There is no need for a reference count since the constants disappear in a stackwise manner, last in first out.

There are however some problems with the non-logical features of Prolog : i.e. the assert and the cut. The effect of the assert is that a temporary constant can become a permanent one. It is not possible to shift the constant at the time of the assert from the temporary constant table to the permanent one, since the constant can be referenced by other database variables from later queries, these constants as we said are not duplicated for the sake of efficiency of the unification. Therefore we propose to only shift the constant from the temporary table to the permanent one on backtracking, when normally the constant would disappear. Then however a reference list has to be constructed for each constant used in an assert in order to change the references when shifting the constant. One can see that a same mechanism has to be provided for builtin predicates like bagof or setof.

A stack frame on the temporary constant table corresponds with a database query and is referenced by a corresponding choicepoint on the run-time stack of the interpreter. On encountering a 'cut' the Prolog interpreter normally removes a certain number of choicepoints from the stack. This however is not possible for database choicepoints : on backtracking the temporary cons ant table has to be cleaned up, using the information which resides in the choicepoint. One can see that only the last (i.e. the oldest) database choicepoint is needed. A set of database choicepoints can then be replaced by a single dummy clean-up choicepoint.

The disadvantage of the temporary constanttable is the complexity of

the implementation and a slight overhead for the 'cut'-operation. The advantage is that the constant table is kept cleaner and garbage collection is not needed frequently.

## 6. Conclusion.

We described the different ways of transforming a Prolog program in a conjunctions of database calls and the mechanisms to integrate a Prolog system and an existing database system effectively. We gave some comments how the interface to the database system could be realized at the Prolog side.

The minimal requirement for a database system (or database machine) to be connectable to the Prolog system enhanced with a database interface as described above, is that it is either an Open system or that we have access to the source code (and a minimal assistance or documentation in order to apply the desired modifications).

This interface has been implemented in our Prolog system. The connexion with Ingres and Unify has been realized.

## 7. Acknowledgements.

This research has been done partly under contract KBAR/SOFT/1 (a grant from De Diensten voor de Programmatie van het Wetenschapsbeleid) and partly in the context of the ESPRIT Pilot Project 107 (sponsored by the Commission of European Communities). The author wishes to thank Maurice Bruynooghe, Bart Demoen, Jose Cotta, Gerda Janssens, Horst Adler and John Gallagher for their useful comments on previous drafts of this paper.

## 8. References.

<Cha82> Chakravarthy, U.S., Minker, J., and Tran, C., Interfacing predicate logic languages and relational databases, Proc. First Int. Logic Programming Conference, Marseille 1982.

<Gal78> Gallaire, H., and Minker, J., Eds., Logic and Data Bases, Plenum Press, New-York 1978.

<Gal83> Gallaire, H., Minker, J., and Nicolas, J.M., Logic and Databases : an overview and survey, Report CERT/CGE, University of Maryland, 1983.

<Ham80> Hammer, M., and Zdonik, S., Knowledge based query processing, Proc. 6th Int. Conf. on very large Databases, Montreal 1980.

<Hen84> Henschen, L. and Shanim, A.N., On compiling queries in recursive first-order databases, Journal of the Association for Computing Machinery, Vol. 31, No. 1, 1984

<Kin81> King, J., Quist : a system for semantic query optimisation in relational databases, Proc. 7th Int. Conf. on very large Databases, Cannes 1981.

<Kom81> Komorowski, H.J., A specification of an abstract Prolog machine and its application to artial evaluation, Linkoping studies in Science and Technology Dissertations, No. 69, Software Systems Research Center, Linkoping University, 1981.

<Loy83> Lloyd, J.W., An introduction to deductive database systems, The Australian Computer Journal, Vol. 15, No. 2, May 1983.

<Ram83> Ramamohanarao, K., Lloyd, J.W., and Thom, J.A., Partial match

retrieval using hashing and descriptors. ACM Transactions on Database systems. Vol. 8, No. 8, 1983

<Rie81> van de Riet, R.P., Wasserman, A.I., Kerstens, M.L., and de Jonghe, W., High Level Programming Features improving the Efficiency of a Relational Database System, ACM Transactions on Database Systems, Vol. 6, No. 3, 1981.

<Ven81> Venken, R., A simple relational database as an extension for Prolog, (in Dutch), Undergraduate Student Thesis, June 81, K.U.Leuven.

<Ven84> Venken, R., A Prolog meta-interpreter for partial evaluation and its application to source to source transformation and query optimisation, Proceedings of ECAI '84, september 1984, Pisa.

<War81> Warren, D.H.D., Efficient processing of interactive relational database queries expressed in logic, Proc. 7th Int. Conf. on very large Databases, Cannes 1981.