A Parallel Multiprocessor Machine Dedicated to Relational and Deductive Databases

R. Gonzalez-Rubio(1), M. Couprie(2)

(1) Bull S.A. Centre de Recherche, DSG/CRG/DAMIA LV 58B1 68 route de Versailles
78430 Louveciennes France Tel : 39 02 51 28 Telex : 697030 F
Electronic mail: Ruben.Gonzalezrubio@crg.bull.fr
(2) Ecole Supérieure d'Ingénieurs en Electronique et Electrotechnique
B.P. 99 93162 Noisy-Le-Grand Cedex

Abstract - Efficiency in databases is a major requirement. This paper presents some solutions to cope with this problem. One solution is to execute operations in parallel: this is done in the "Delta Driven Computer" DDC, which is a multiprocessor machine with distributed memory dedicated to relational and deductive databases. In DDC, relations are distributed among the nodes of the machine, and the data are processed asynchronously in each node. To do that in an efficient way, a coprocessor, specialized for relational operations, is also proposed. It is called μSyC, for "microprogrammable Symbolic Coprocessor". This paper is divided into two parts. The first part describes DDC, presenting the architecture, the languages, and an original computational model. The second part describes μSyC, its architecture, instruction set and the data structures used at the μSyC level.

Introduction

Efficiency in databases is a major requirement. Two complementary solutions are proposed here to cope with this problem. One solution is to increase efficiency through parallelism. The "Delta Driven Computer" DDC[1] is a multiprocessor machine with distributed memory dedicated to relational and deductive databases. The second solution is to accelerate the local processing in each processor node of the machine. This is done by the "microprogrammable Symbolic Coprocessor" μSyC[2] .

DDC is currently under development at BULL Research Center. The DDC project includes μSyC's design and realization as a sub-project. Early papers describe some of our ideas about DDC [Gon 86] and about μSyC [Cou 87]. An overview of the DDC project is presented in [Gon 87].

The architecture of DDC was kept simple: it consists in a multiprocessor system composed of a set of interconnected PCM (Processor, Communication and Memory) nodes, with distributed memory. The originality of our work is the way parallelism is achieved. From a conceptual point of view, DDC executes a language based on production rules, called VIM (Virtual Inference Machine), where parallelism is implicit. The execution of a VIM program is performed using a forward chaining strategy. Given a set of rules and a set of initial facts, the only operation mode of DDC is saturation, that means that all the conclusions are found.

[1] This project is partially supported by **ESPRIT-415**.
[2] This project is partially supported by **ESPRIT-415** and **ESPRIT-956**.

Parallelism in the machine is achieved by distributing the facts (tuples) among the PCM nodes and by firing rules independently in each processor.

VIM is an intermediate language; so, in the project, we are working on the compilation of a high level language into this intermediate language. The high level languages which we are considering are declarative ones (e.g. SQL and Logic Programming).

VIM execution is possible using the DDEM (Delta Driven Execution Model). In this model, execution is driven by the facts deduced from the rules. We call such a fact a Δ (Delta). The parallel architecture we propose can support DDEM.

The implementation of DDEM is based on relational operations, so VIM rules are transformed into a DDCL program (Delta Driven Computer Language).The DDCL level is μSyC's level of operation.

The choice of the data structures is crucial to ensure the efficiency of the microprogrammed algorithms. Since we use a finite state automaton based representation for the relations stored in memory, and because this type of representation can be very space-consuming in its naive form, we use compaction techniques derived from those described in [Roh 80].

The design of μSyC was based on the study of the algorithms and data structures used in DDC. Since μSyC is microprogrammable, this instruction set can be modified to accept new functionalities. The circuit can also be used for other similar applications, like data filtering from disk or from memory. A set of microprograms has already been written for this application [Cou 87] using algorithms very similar to those of the SCHUSS filter [Gon 84].

A performance evaluation showed that μSyC is about five times faster than the MC 68020 with a 60 ns cycle time, while executing DDC operations.

In the first part of this paper, we describe the architecture of DDC, its computational model, and the different language levels from SQL or Datalog down to the machine level language called DDCL.

In the second part, we describe the aims of μSyC, its architecture and specialized operators, and its instruction set used to implement DDCL. We also describe and evaluate the data structures that we use at the μSyC level.

Part I: DDC

I-1 Motivations

The basic aim of the DDC project is to design an efficient computer dedicated mainly to databases and symbolic computation, for large "knowledge based" applications.

We decided to study a parallel architecture in order to attain high performance. But a drawback of parallel systems is the cost of software developments. Maintaining software costs as low as possible could be achieved by using a simple high level language where parallelism is hidden from the programmer.

A prerequisite to the design of a specialized parallel architecture is an execution model well adapted to the application domain and able to handle parallelism in a simple way. Another prerequisite is that the architecture must not be dedicated to one particular language. Instead, it must be flexible enough to accommodate a variety of declarative programming styles, including relational [Ull 82] + deduction [Gal 78], logic [Kow 79], and functional [Bac 78]. Typical applications for DDC stem from relational databases, deductive databases, expert systems, simulation systems, etc.

I-2 The architecture

The architecture of DDC consists of a set of PCM nodes (Processor, Communication device, Memory), an interconnection network and a processor acting as a front-end (Host). There is no need for a shared memory (fig. I-1).

The processor of a PCM node has two parts, one general purpose microprocessor, a MC68020 and one symbolic coprocessor μSyC. The latter is a custom VLSI which works as a MC68020 coprocessor and is specially used in DDC to execute some operations faster than the processor. Memory is built with "off the shelf" chips. The communication module of a PCM node sends and receives messages to and from the interconnection network.

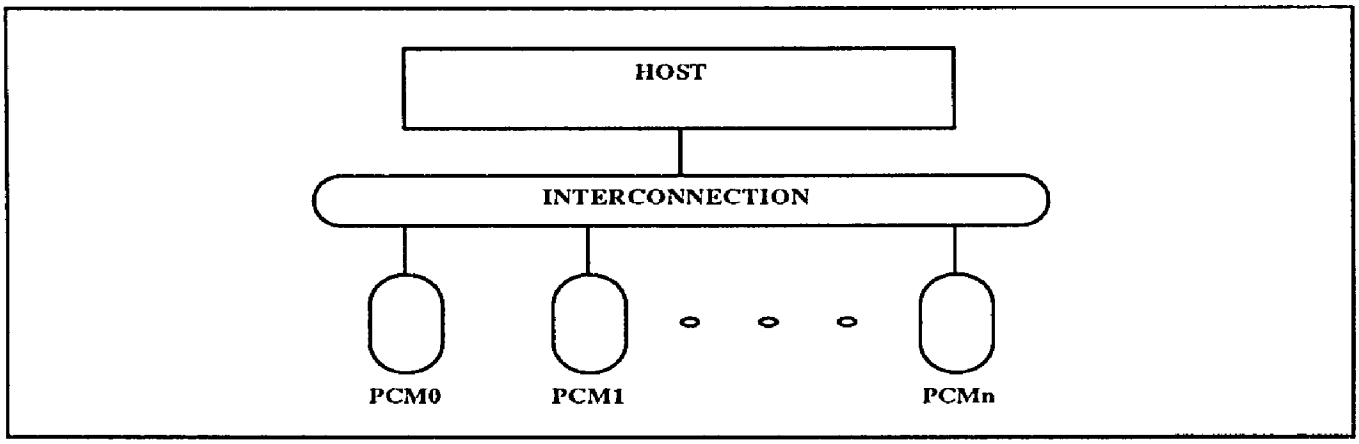

Figure I-1: DDC architecture

I-3 Languages on DDC

As development methodology, we described all the levels, from the high level language down to the machine level language, and how to go down from one level to the next one (propositions of this method can be found in [Boy 78], [Das 84]).

The language levels in DDC are:

- a high level language, derived from relational technology, logic programming or functional programming. Currently, we are only considering the database and logic programming aspects. This language can be SQL [Dat 87], for the compatibility with other relational systems. An other language can be Datalog, as introduced in [Gal 78], [Ull 85] and [Ban 86], to handle deduction in the database.
- an intermediate language is the key of our approach. This language is based on additive production rules with a forward chaining (saturation) strategy. We call this language VIM (Virtual Inference Machine). Some principles of production rule systems are presented in [Ver 77], [Wat 78], [For 79].
- to execute the saturation, the Delta Driven Execution Model (DDEM) is provided. This model can be parallelized, as explained below. In fact a VIM program is translated into a DDCL (Delta Driven Computer Language) program. DDCL is the DDC machine language and the program is executed following the DDEM.

The translation from a logic programming language into a VIM program is done following an algorithm called the Alexander Method, proposed by J. Rohmer [Roh 85, 86]. The algorithm is also described in [Ker 87]. The Alexander Method is an algorithm which transforms a set of VIM rules and a query into a new set of focalized rules. The Alexander Method simulates backward chaining using forward chaining.

I-4 The Virtual Inference Machine VIM

The design of VIM is based on our previous experience with production systems. Basically, this language is composed of production rules, i.e. rules of the form:

$h_1 \Rightarrow c_1, \ldots\ldots, c_p$
$h_1, h_2 \Rightarrow c_1, \ldots\ldots c_p$
$h_1, \neg h_2 \Rightarrow c_1, \ldots\ldots c_p$

where the h_i and c_j are predicates of the form: $p(X_1, \ldots\ldots, X_n)$ and where X_i is either an atom (constant) or a variable. This means that functions (or trees) are not visible at this level.

We require that the variables in the conclusions also appear in the hypotheses (note that the

restrictions we impose on VIM are the same as those of Datalog). This means that if initially there exists a set of facts containing only constants, all the generated facts will also contain only constants.

The only operation mode of the machine is saturation: given a set of rules and a set of initial facts, find all the possible conclusions. In a VIM program, it is possible to execute a sequence of saturations. In this case, we call each saturation a strate. Stratification [Apt 86] is useful to deal with negation problems.

I-5 The Delta Driven Execution Model DDEM

DDEM is a model to execute saturation on a set of VIM rules. It is possible to describe the execution model in an informal way at the VIM level. When a rule is applied, a fact or a set of facts may be produced. Each produced fact is called a BΔ (read Black Delta). Only those facts which are not already present in the database are considered as new facts: we call them WΔ (read White Delta). A WΔ can be inserted in the database, and it can be used to trigger the execution of rules.

For example, a VIM rule like: p,q => r is transformed into two delta-rules:

$W\Delta p,q_c \Rightarrow B\Delta r$ and $W\Delta q,p_c \Rightarrow B\Delta r$

where q_c and p_c are the current representations of q and p in the database.

BΔr may already exist in the database: if it is the case, it is called a duplicate, and nothing more happens. If it is not, the BΔ becomes a WΔ. WΔr contains a new fact on r that has just been produced. Consequently, this WΔr is added to the current representation of r in the database, and then the whole set of rules is tried again using just the WΔr as trigger in rules of the form:

WΔr,... => BΔ...

When no more WΔ is produced the logic database is saturated, then the processing is over.

Two types of process have been specified, the application of rules and the elimination of duplicates (fig. I-2). In the case where rules are monotonic and commutative the order of Δ arrivals does not modify the final result. This means that the model is asynchronous and parallelism is implicit.

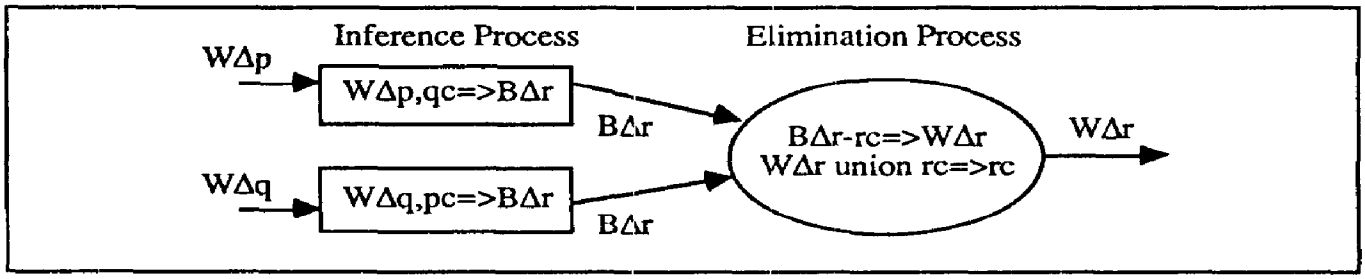

Fig I-2: DDC process

I-6 From VIM to DDCL

A predicate in the VIM environment corresponds to a relation in the DDCL environment. Thus, fact and tuple are two representations of the same semantic element (fig. I-3).

At the DDCL level, a Δ is a tuple and with each Δ, a set of primitives is associated which has to be executed on it. The piece of DDCL code that contains this set of primitives is a special entity called an action, whose identifier is closely related to the Δ. An action contains primitives which correspond, at VIM level, to the rules where the associated Δ is to be used (fig. I-4).

I-7 The Delta Driven Computer Language DDCL

DDCL is the language of the machine. Its semantical power is the one of μSyC operations.

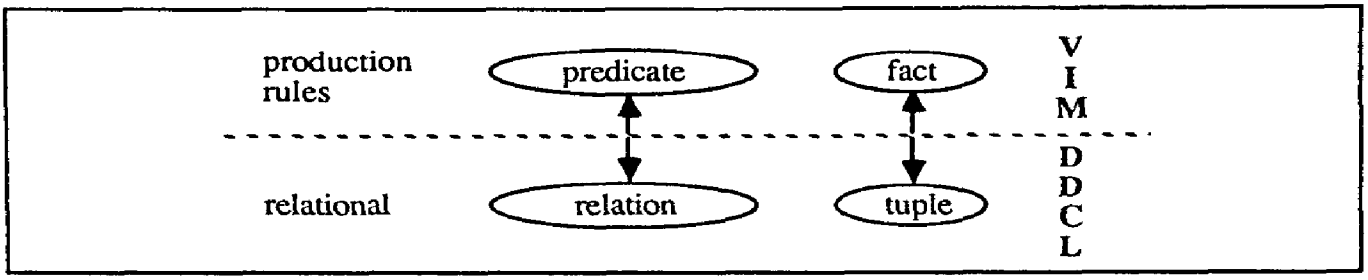

Fig I-3: VIM and DDC worlds

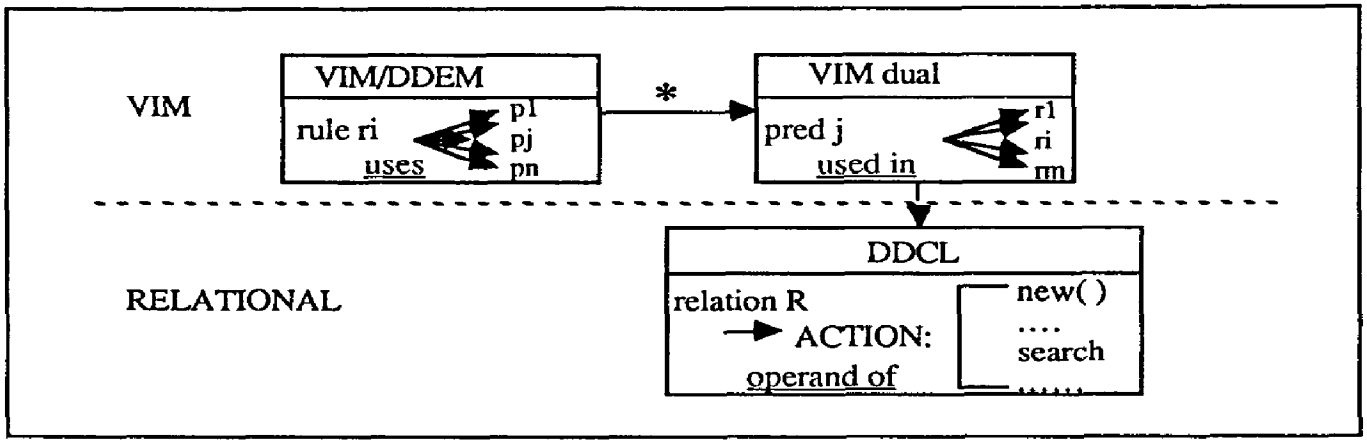

Fig I-4: Transforming Rules into actions

A DDCL program is structured into several independent modules, each corresponding to the code of an action.

There are four different types of DDCL primitives: program control primitives, operations on relations, "propagation" operations (communication among processors), and input/output primitives.

DDCL consists mainly in operations on relations. Most of them are executed following the principles of filtering by automata presented in [Gon 84, 87b], and the join algorithm LA-JOIN presented in [Bra 86].

The DDCL primitives are:

- the control primitives for conditions or loops *(if..then..endif, with..map..endmap, with..mapfile..endmapfile)*;
- the input/output primitives (*access_tuple, write*);
- the propagation primitives to create and send a delta to a destination node *(message)*. The destination is calculated according to the hash function applied to the attribute values.
- the operations on relations, which include data updating and pattern matching:
  - ***add*** adds a tuple in the automaton representation of a relation;
  - ***new_add*** first checks if a tuple exists in the automaton representation of a relation. If it is the case, new_add returns the boolean value ***false***, if not, new_add adds the tuple to the representation and returns ***true***;
  - ***search*** retrieves part of a tuple in the automaton representation of a relation. Whenever the matching phase succeeds, the final pointer stored in the automaton (which points to the list of possible ends of tuples) is returned to permit the creation of solution tuples.

I-8 The mapping strategy

The mapping strategy of DDEM into the DDC architecture is statically determined and dynamically executed. We try to balance the load in the machine dynamically, while minimizing the communications. So at any moment during a saturation process, we can assume that the facts are distributed among the PCM nodes and all the compiled rules are copied in each node.

A relation is distributed according to a hash code function applied to the value of one of its arguments. This approach is similar to the notion of buckets introduced in [Bra 84], [Kit 83] for the representation of relations, where the partitioned buckets represent disjoint subsets of the original relation and provide a natural basis for parallelism (also mentioned in [DeW 86]). Predicates can be duplicated. If so, the copies are distinguished at compile time by adding a suffix to the predicate name.

The advantage of this mapping is that data are sent towards the only node where they can possibly be used. This improves the locality factor and reduces the rate of communications.

Consider a DDC composed of three nodes, a set of two VIM rules:

R_1: p(X,Y), q(Y,Z) => r(X,Z)
R_2: p(X,Y), r(X,Z) => p(Y,Z)

and four initial facts:

p(aa,bb), q(cc,dd), q(bb,cc), r(aa,cc)

These facts must be stored as relations as follows: at compile time, it can be determined that the relation p is used in R_1 and R_2 but in each one according to different attributes. So, in the machine, instead of having p , there are two copies: p_{c1} and p_{c2}.

$p_{c1}(X,Y)$, to be used by R_2 , is distributed according to values of its first argument.
$p_{c2}(X,Y)$, to be used by R_1 , is distributed according to values of its second argument.
For q there is just q_{c1} distributed according to values of its first argument.
For r there is just r_{c1}.

So if we apply the hash function to values of arguments of the initial facts, we can identify in which processor a tuple will be stored:

$H(1, p_{c1}(aa,bb)) = H(1, r_{c1}(aa,cc)) = h(aa) = 2$
$H(2, p_{c2}(aa,bb)) = H(1, q_{c1}(bb,cc)) = h(bb) = 1$
$H(1, q_{c1}(cc,dd)) = h(cc) = 3$

where H is a function, which takes as arguments: a tuple, and the rank of the attribute to which the function should be applied; h is the hash function which takes as argument the value of the selected attribute, and returns the identifier of the destination node; and 1, 2, 3 are the PCM node numbers (fig. I-5).

The hash function h is applied to each tuple of a BΔ produced at a given node. Then each tuple is sent to just one processor.

Note that the execution is made independently on each node. As soon as a node receives a Δ, it can execute an action and possibly produce some new Δs that will activate other nodes or maintain this node in activity.

I-9 An example of deduction in DDC

Let us have the VIM rules:

R1: father(X,Y) => ancestor(X,Y)
R2: ancestor(X,Y),ancestor(Y,Z) => ancestor(X,Z)

The database contains a relation named father.

There are two ways to calculate the descendants of "Jean". The first one is to add the rule:

R3: ancestor("Jean",X) => solution (X)

DDC will deduce or produce in forward chaining all the facts "ancestor", and those which are in the solution will be selected on the fly.

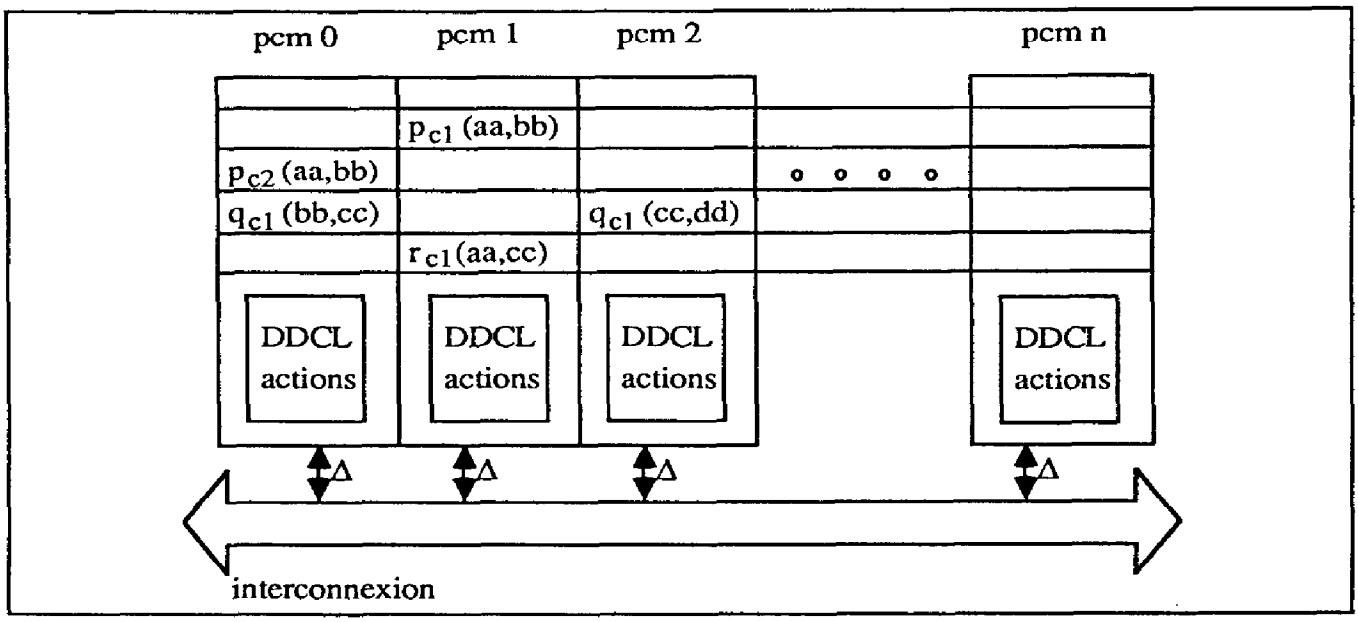

Figure I-5: Mapping DDEM into DDC

The second way is the application of the Alexander Method.

In both cases, DDC receives a set of VIM rules to execute in forward chaining. When the Alexander Method is applied, an important speed-up is obtained: instead of producing all the ancestors, only the ancestors that are necessary to compute the solution are produced. It may be seen as forward chaining emulating backward chaining.

Now, let us see how the three VIM rules in our first example are compiled into DDCL code. First, the rules are rewritten as:

R1': INPUT "file_father" (X,Y) => ancestor(X,Y)
R2: ancestor(X,Y),ancestor(Y,Z) => ancestor(X,Z)
R3': ancestor("Jean",X) => OUTPUT "file_solution" (X)

A DDCL program has the following structure:

MAIN <set_of_actions 1> NODES <set_of_actions 2> END

<set_of_actions 1> is executed by the host computer, while *<set_of_actions 2>* is copied into every node. Then, each node executes some actions depending of the Δ arrivals.

The "ancestor" relation is duplicated. rel_ancestor_1 is distributed among the nodes by the application of the hash function to the first attribute, and rel_ancestor_2 is distributed by the application of the hash function to the second attribute. rel_ancestor_1 is used as the reference for the elimination of the duplicates.

The resulting DDCL program is:

```
MAIN                                    /* Beginning of the host actions        */

action saturation :                     /* This action opens files, and loads the   */
    message(all,init,<>)                /* source relation into the machine's nodes */
    open("file_solution",solution,write)
    open("file_father",father,read)
    with file(father) mapfile
        access_tuple(<X,Y>)
        message(node,ancestor,<X,Y>)
```

```
            if X="Jean" then
                message(node,ancestor_1,<Y>)
            endif
        endmapfile
        close(father)
        termination
endaction

action output_1 :                          /* Stores the Δs arriving to the host into the  */
        access_delta(<X>)                  /* solution file                               */
        write(solution,<X>)
endaction

action finish :                            /* Closes the solution file                    */
        close(solution)
endaction

NODES                                      /* Beginning of the nodes actions              */

action init :                              /* Initializes the local relations in each node */
        idle                               /* and initializes the termination action      */
        init_relation(rel_ancestor_1)
        init_relation(rel_ancestor_2)
        init_relation(rel_ancestor_sol)
        init_termination(finish)
endaction

action ancestor_1 :                        /* Elimination of duplicates and first part    */
        access_delta(<X,Y>)                /* of the Δ-rule                               */
        if new_add(rel_ancestor_1,<X,Y>) then
            message(node,ancestor_2,<Y,X>) /* a WΔ */
            with search(rel_ancestor_2,<X>) map
                access_tuple(<Z>)
                message(node,ancestor_1,<Z,Y>) /* a BΔ */
                if Z="Jean" then
                    message(node,ancestor_sol,<Y>) /* a BΔ */
                endif
            endmap
        endif
endaction

action ancestor_2 :                        /* Dual part of the Δ-rule                     */
        acces_delta(<X,Y>)
        add(rel_ancestor_2,<X>)
        with search(rel_ancestor_1,<X>) map
            access_tuple(<Z>)
            message(node,ancestor_1,<Y,Z>) /* a BΔ */
            if Y="Jean" then
                message(node,ancestor_sol,<Z>) /* a BΔ */
            endif
        endmap
endaction

action ancestor_sol :                      /* Eliminates the duplicates in the solution   */
        access_delta(<X>)                  /* and sends Δs to the host                    */
        if new_add(rel_ancestor_sol,<X>) then
            message(main,output_1,<X>)     /* a WΔ */
        endif
endaction

END
```

I-10 Example of relational operations in DDC

Let us consider a relation S with the attributes: S#, SNAME, STATUS, CITY.

A selection query can be expressed by the VIM rule:
rel_S(X,_,Y,"Paris") => solution(X,Y)

Let us now consider the relation S and a relation SP with the attributes: S#, P#, QTY.

The following VIM rule is an example of join operation:
rel_S(X,_,_,Y),rel_SP(X,Z,_) => solution (Z,Y)

These two examples show how these queries can be translated into VIM rules, in order to be executed in DDC. The translation of other relational operations to VIM is possible: VIM is relationally complete because it can express the five basic relational operations. Because VIM can also express recursion, DDC is a relational deductive system.

Part II: μSyC in DDC

In every node of DDC, the 68020 processor will be assisted by μSyC (microprogrammable Symbolic Coprocessor). μSyC is microprogrammable because we want to have an adaptable instruction set, which corresponds to DDCL and its evolution.

We have studied μSyC speed (with a cycle of 100 ns) compared to a 68020 (with a cycle of 60 ns). The simulation showed that μSyC is about five times faster, for our application, than the MC68020.

The design of μSyC is now finished. It is currently being implemented with a HCMOS 1.2 micron, standard cells technology. μSyC will be integrated to the DDC prototype in 1989.

II-1 Motivations

The most basic data in databases are character strings and numbers (integers, reals). While numbers are easily and naturally processed by computers, character strings are more difficult to compute because of their variable length. One of the most frequent basic operations in DBMSs consists in searching whether a given character string can be found or not within a huge set of strings. This operation is in fact, one of the primitives of DDCL, the "search" primitive.

Relations must be stored in memory in a structured way. To facilitate searches, we have chosen to store some attributes of a relation as a compacted automaton. Traditionally, the representation of an automaton in memory consists in a two-dimensional array, where each row represents a state of the automaton. In the columns are stored pointers to other states, which represent the transitions between states. When using the automaton, the input character indexes the current state or row in the array, to find a pointer to the new current state.

The problem with this representation of automata is that in most cases the array is very large, and contains mostly null pointers, assuming that a null pointer indicates the failure of the search in the automaton.

Instead of this space-consuming representation of automata, we use a compact format with several types of state representation. The choice of a representation for a given state will depend on the number of transitions starting from this state.

This is why every pointer to an automaton state must be tagged with the type of the state's representation, so that the search program can know how to compute the next state when an input is given.

II-2 μSyC architecture

μSyC may be seen by the host processor (the 68020) as a DMA-type peripheral, that is, μSyC may request the mastership of the bus. The external microprogram memory is accessed via a

12-bit micro-address bus and a 69-bit micro-instruction bus. As a matter of fact, it is possible, thanks to the 69-bit micro-instruction word, to command 1 to 8 simultaneous actions, e.g. two arithmetic operations, two byte extractions with the decrement of associated counters, a byte comparison and a memory access request.

As shown in fig. II-1, µSyC is composed of a data path, a micro-instruction sequencer, a memory controller, and host CPU interface logic and registers.

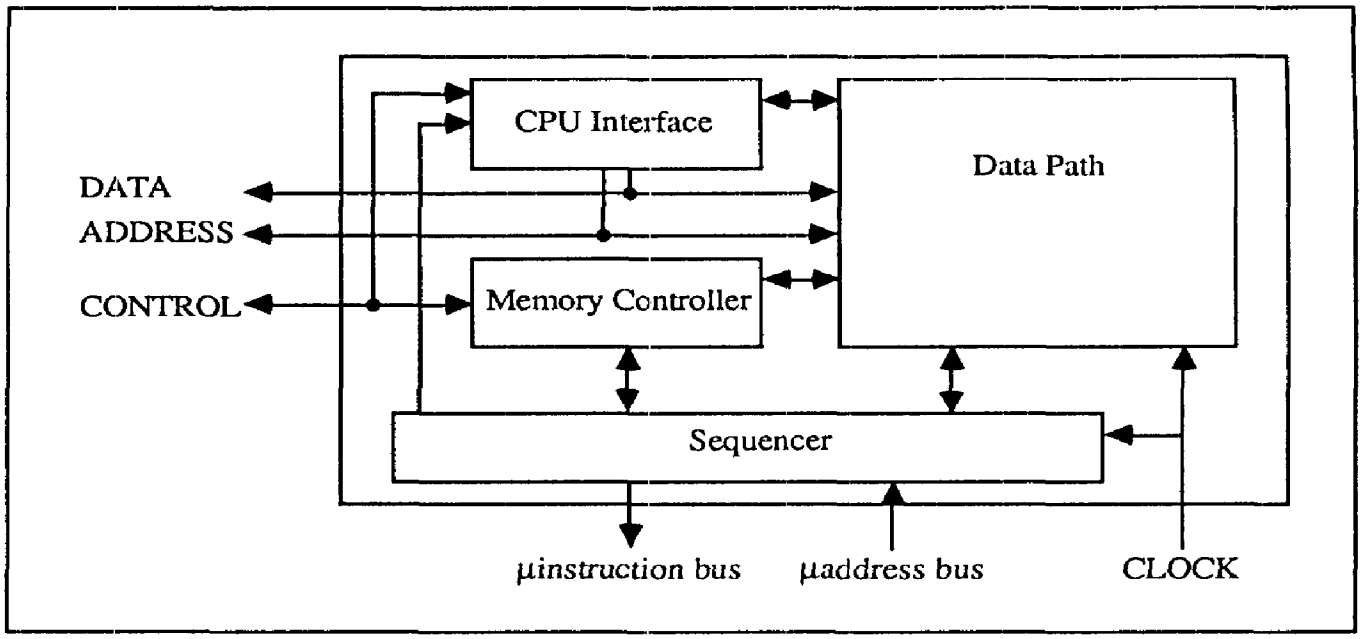

Figure II-1: µSyC internal organization

The data path is organized around two main busses, the data bus and the address bus. There are smaller secondary busses for the connection between the main busses and independent operators like counters and byte extractors.

The data bus connects together sixteen 32-bit registers, one ALU, one shifter, a byte injector, a bit tester that can test a bit among 32, and a set of four wired 32-bit constants.

The address bus connects together one ALU, sixteen 32-bit registers and four 32-bit wired constants. In addition, there is a tag comparator and a 4-bit three-state gate that allows a tag to be associated to a 28-bit pointer.

The other operators are two 1-byte extractors, one byte comparator and two sets of counters. Each counter set is made up of four registers and a decrementer. These counters can drive directly the byte extractors and the byte injector, giving the position in which the byte is to be extracted from or inserted into a 32-bit word.

The sequencer of µSyC contains an eight 12-bit address stack for sub-microprogram calls, an 8-bit counter for short loops, and an address calculator able to perform multi-destination branches and calls in a single cycle.

The memory controller allows the microprogrammer to ignore such details as the number of cycles the memory needs to perform a read operation, wait states, bus errors, etc.

The host CPU and µSyC communicate via 9 interface registers and 9 control signals. All of the interface registers are read/write for both µSyC and the host CPU. One of the registers serves as a status word and a particular bit in this word is used to start µSyC at the beginning.

Specialized operators in µSyC

µSyC has specialized hardware features in order to deal efficiently with character strings and

tagged pointers. These operators are in the data path. μSyC has a 32-bit data bus and a 32-bit address bus. In order to minimize the number of memory accesses, μSyC only reads or writes 32-bit words, but it has specialized operators able to extract one byte from a 32-bit word, or insert a byte into a 32-bit word, at any 8-bit aligned position.

There are two byte extractors (with their associated registers) and one byte comparator, that can operate in parallel. These features make character string operation - especially comparison - very efficient in μSyC. One byte injector is used to prepare the results before writing them - 32 bits at a time - back into memory.

μSyC also has specialized hardware features that make tag manipulation easy and efficient. Tags are 4 bits long and are located in the 4 higher-order bits of a 32 bit word. Thus, 16 different type identifiers can be associated with pointers, while 28 bits remain available to address a 256 megabyte memory space.

II-3 μSyC's instruction set for DDC

When a delta reaches a node, its tuples are immediately stored into the local memory of the node, and pointers to these tuples are pushed onto a stack. This work is done by the 68020. Then, the 68020 triggers μSyC and lets it work on the data, assuming μSyC has a program indicating what to do with this data, and knows the location where the tuples are referenced.

The three principal actions that μSyC is asked to perform are ***search***, ***add*** and ***new_add***. μSyC must also know how to execute a very simple stack-based language, allowing the scheduling of the operations on tuples and relations. At the end of its program, or during program execution, μSyC will need to send messages either to other nodes or to the host system.

The search, add and new_add instructions

Their implementation depends heavily on the format under which the relations are stored in memory. Since we have chosen a compact finite state automaton format for the relations, the search instruction is very fast indeed, the number of μSyC cycles it lasts is proportional to the length of the tuple.

Messages

Sending a message is not a job that μSyC can handle easily, because it deals with system features and routines that are written in 68020 machine code. Thus, the message primitives of μSyC are only requests to the 68020, and only after a message request has been answered, can μSyC start its work again.

A general purpose stack language

In order to schedule these operations and to do some elementary calculus, we defined a low level general purpose stack-based language. This simple language is necessary to implement the high-level control structures of DDCL ***(if.. then.. endif, with.. map.. endmap, with.. mapfile.. endmapfile)***.

II-4 The data internal format

Finite state automata are used here to retrieve very quickly a tuple from a relation, given a key attribute or a set of keys. Since relations are large, we use compaction techniques in order to reduce the size of the automaton representation in memory. These techniques must not penalize too much the performance of the search operation, while reducing significantly the memory usage. For each state, the appropriate format is chosen dynamically at creation time depending on the number of transitions.

The vector state representation

An automaton state may be represented as a N-element vector, N being the size of the alphabet.

The rank of an element in the vector corresponds to the input character code, and the value of this element indicates the transition induced by this input character. The element in the vector is null if there is no transition in this state for this input, and otherwise contains the address of the next state (fig. II-2).

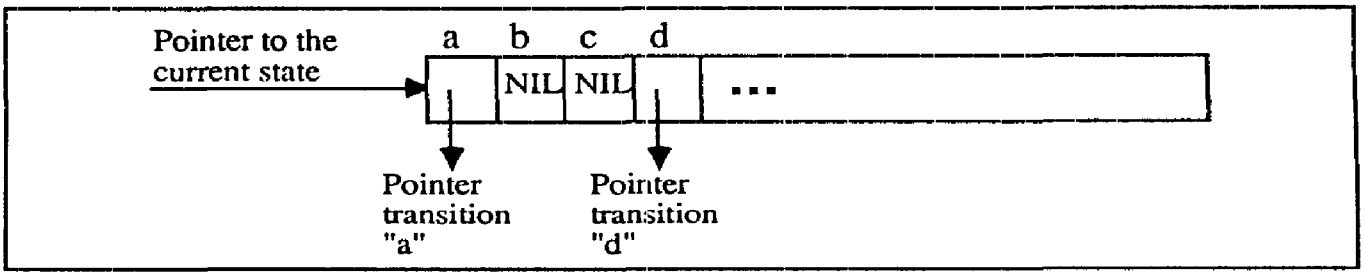

Figure II-2: The vector representation

This representation is very expensive in terms of memory space, and needs to be initialized, but it allows a very fast searching, because there is only one indexed memory access in the transition program. It is used when a state has many transitions.

The linear representation for states

Sequences of states with a single transition in each state are very common in a database context. We call such a sequence a "linear" state sequence, and it can be represented by a simple character string (fig II-3), allowing the transitions between states to be simple and fast (a comparison and the increment of a pointer).

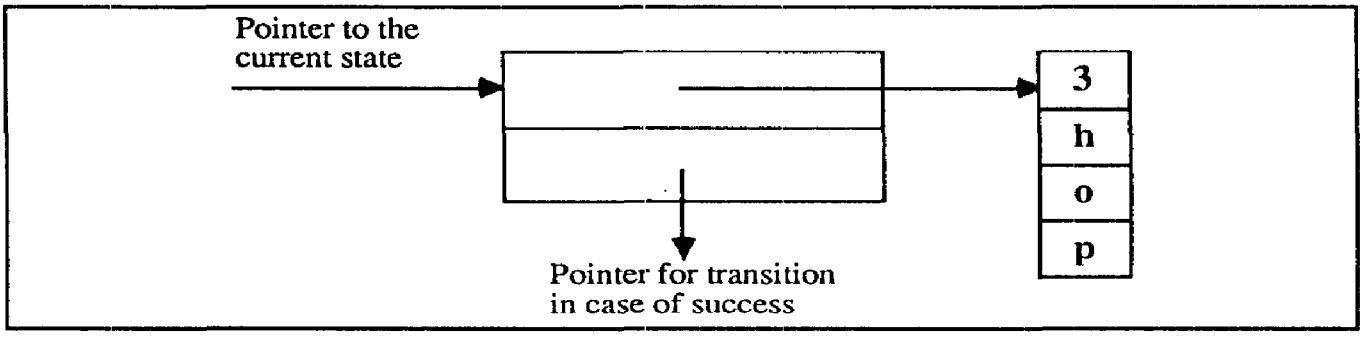

Figure II-3: The linear representation

The packed state representation

A state with a small number of transitions (e. g. 2 to 6) may be represented as a list of couples (A_i, P_i) , where A_i is a value within the alphabet and P_i is a pointer to the next state if the actual input value equals A_i (fig II-4).

The program calculating the transition for such a state representation sequentially searches if the actual input matches one of the A_i in the state, and sets the new current state pointer to P_i if this is the case. This is of course slower than other transition programs, and the automaton construction program must have a simple garbage collector to manage this representation, but this structure saves a lot of memory space compared to the vector one.

Splitting the alphabet

Further compaction can be achieved by splitting the alphabet into two parts: the "usual characters" which are the most frequent in the database, and the other characters.

The representation of one individual state is divided into two parts. The first part corresponds to the "usual character set", and uses the vector and packed representation. The second part

corresponds to the other characters, and for those characters we only use a list representation like the packed one.

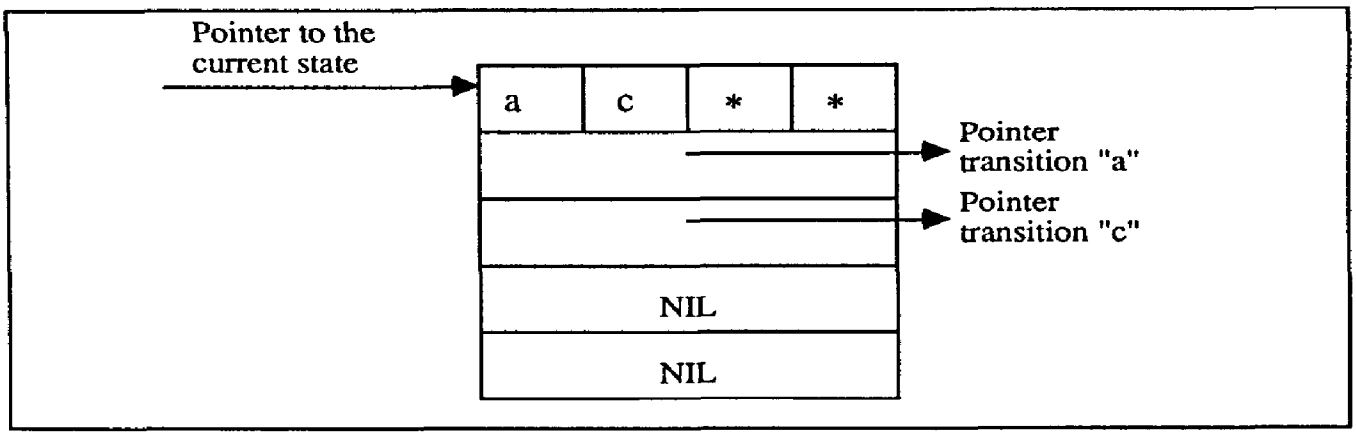

Figure II-4: The packed representation

The drawbacks of this method are the need of a transcoding table, and the fact that the efficiency of the representation depends on some statistical information about the contents of the database.

II-5 An evaluation of the compact automaton format

A pragmatic evaluation of the algorithms based on the previously described data structures shows the interest of this approach. In this section, we present the results of this evaluation, which is concerned with both speed and memory occupation. Further details about this experiment can be found in [Cou 88].

Let Qi be the initial quantity of data, in bytes, which will be represented by the automaton. Let Qf be the final quantity of memory space, in bytes, which is necessary to hold the automaton. We will call Expansion Rate (Ex) the ratio Qf / Qi , and we will focus on the variations of Ex with the use of the different data structures.

We are also interested in the time -more precisely, the number of memory accesses- which is necessary to construct (compile) the automaton and to search a string in the automaton:

Tc = (number of memory accesses for compiling Nc characters) / Nc
Ts = (number of memory accesses for searching a Ns character long string) / Ns

The initial data has been randomly generated, following a Gaussian law. The parameters of the random generator have been statistically determined. 10000 strings were generated for each configuration. The alphabet comprises 256 characters. The size of a pointer is 32 bits.

The following table summarizes the results of the evaluation:

	Vectors	Vectors + Linear	Vectors + Linear + Packed	Vectors + Linear + Packed + Separation
Ex	907	20.8	2.94	1.81
Tc	61.6	4.40	3.82	4.76
Ts	2.00	2.24	2.49	3.41

We point out an important decrease of Tc with the introduction of linear and packed structures. This is due to the decreasing number of vector structures, and therefore to the reduction of initialization time.

These results show that the expansion rate Ex between the size of initial data and the size of the resulting automaton can be reduced down to 1.81, while compilation and searching remain efficient.

Conclusion

Our aim is to design and implement a very fast database machine. We try to increase performance by two ways:

- by using several nodes working in parallel,
- by placing a hardware accelerator in each node of the machine.

We have presented here the architecture of DDC, the different language levels and the execution model. The kind of parallelism exhibited by this model is well suited to database operations in memory.

We also presented µSyC, a VLSI accelerator for relational operations. µSyC executes DDCL primitives faster than the 68020, and it can easily be adapted, by microprogramming, to other kind of database or symbolic applications.

The DDC prototype has been running since december 87 on a Bull SPS 7 with 4 processors. The prototype is mainly oriented toward database operations and deductions. Its purpose is to prove that the parallelism handled by the model/machine has a favorable cost/performance ratio. Evaluations are on the way.

The design of µSyC is now finished. It is currently being implemented with a HCMOS, 1.2 micron, standard cells technology. µSyC will be integrated to the DDC prototype at the end of 1988. A second, "custom" version of the circuit is planned for 1990.

Acknowledgements

The following persons have played active roles in the design and implementation of DDC and µSyC: B. Bergsten, A. Bradier, J. Callot, J. Garcia, B. Kerhervé, T. Maréchal, J. Rohmer, D. Terral. We also thank all the students working with us for their help, critical mind and fresh ideas.

References

[Apt 86] Apt K., Blair H., Walker A.: "Towards a Theory of Declarative Knowledge". Proc. Workshop on Foundations of Deductive Databases and Logic Programming, Washington D.C., pp 546-629, 1988.

[Bac 78] Backus J. "Can Programming be liberated from the Von Neumann style? A Functional Style and its Algebra of Programs". CACM, vol 21., no.8, pp 613-641, aug.1978.

[Ban 86] Bancilhon F., Ramakrishnan R.: "An amateur's introduction to Recursive Query Processing Strategies". Proc. of the ACM SIGACT-SIGMOD Symp. on Princ. of Database Systems, 1986.

[Bra 84] Bratbergsengen K.: "Hashing Methods and Relational Algebra Operations". Proceedings of the 1984 Very Large Database Conference. aug.84.

[Bra 86] Bradier A.: "LA-JOIN: Un Algorithme de Jonction en Mémoire et sa Mise en Oeuvre sur le Filtre SCHUSS. II^{èmes} Journées Bases de Données Avancées. Giens, apr. 1986.

[Boy 78] Boyd D.L., Pizzarello A.: 'An Introduction to the WELLMADE design methodology". IEEE Trans.Soft.Eng, TSE. 4,4. Jul.1978, pp 276-282.

[Cou 87] Couprie M., Garcia J., Maréchal T., Terral D.: "µSyC: Coprocesseur Microprogrammable pour les Applications Symboliques". Journées Firftech Bases de Données et Intelligence Artificielle. Paris, apr. 87.

[Cou 88] Couprie M.: "Présentation et évaluation d'algorithmes pour les bases de données en mémoire basés sur la notion d'automate d'états fini". Bull Research Center, report DGS/CRG/88005, febr. 1988.

[Das 84] Dasgupta S.: "The Design and Description of Computer Architectures".Eds. Wiley-Interscience, 1984.

[Dat 87] Date C.J.: "A Guide to the SQL Standard". Addison-Wesley. 1987.

[DeW 86] DeWitt D.J., Gerber R.H., Graefe G., M. L., Kumar K. B., Muralikrishna: "GAMMA - A High Performance Dataflow Database Machine". 86 VLDB Conference, Kyoto, Japan, aug. 1986.

[For 79] Forgy C.L.: "On the Efficient Implementation of Production Systems". Ph.D. at Carnegie-Mellon University, febr. 1979.

[Gal 78] Gallaire H., Minker J. (eds.): "Logic and Databases".Plenum, New York 1978.

[Gon 84] Gonzalez-Rubio R., Rohmer J., Terral D.: "The SCHUSS Filter: A Processor for Non-Numerical Data Processing". 11th Annual International Symposium on Computer Architecture. Ann Arbor. 1984.

[Gon 86] Gonzalez-Rubio R., Bradier A., Rohmer J.: "DDC Delta Driven Computer. A Parallel Machine for Symbolic Processing". ESPRIT Summer School on Future Parallel Computers. University of Pisa, june 1986.

[Gon 87] Gonzalez-Rubio R., Rohmer J., Bradier A.: "An overview of DDC: Delta Driven Computer". Conference on Parallel Architectures and Languages Europe. Eindhoven, june 1987.

[Gon 87b] Gonzalez-Rubio R.: "Propositions d'Architectures pour les Traitements Symboliques" Thèse d'Etat. Université Pierre et Marie Curie Paris VI, Paris, 1987.

[Ker 87] Kérisit J.M.: "A Relational Approach to Logic Programming: The Extended Alexander Method". Bull Research Center, Technical report DLA/SLIA 87004 febr.87.

[Kit 83] Kitsuregawa M., Tanaka H., Moto-oka T.: "Application of Hash to Database Machine and its Achitecture", New Generation Computing, vol.1, No.1, 1983.

[Kow 79] Kowalski R.A.: "Logic for Problem Solving". Elsevier Sciences Publishing. 1979.

[Roh 80] Rohmer J.: "Machines et Langages pour Traiter les Ensembles de Données". Thèse d'Etat. INPG Grenoble, 1980.

[Roh 85] Rohmer J., Lescoeur R.: "The Alexander Method. A technique for the processing of recursive axioms in deductive databases". Bull Research Center Report 1985.

[Roh 86] Rohmer J., Lescoeur R., Kerisit J.M.: "The Alexander Method. A technique for the processing of recursive axioms in deductive databases". New Generation Computing, 4. 1986.

[Ull 82] Ullman J.D.: "Principles of Database systems". Computer Science Press, Rockville MD. 1982.

[Ull 85] Ullman J.D.: "Implementation of Logical Query Languages for Databases". ACM Trans. on Database System 10(3) pp. 289-321, sept.85.

[Ver 77] Vere S.A.:"Relational Production Systems". Artificial Intelligence pp. 47-688, febr.1977.

[Wat 78] Waterman D.A., Hayes-Roth F.:"An Overview of Pattern-Directed Inference Systems". Pattern Directed Inference Systems, D.A. Waterman and F. Hayes-Roth. Eds., Academic Press, New York, 1978.