Verso: A data base machine based on non 1NF relations

J. Verso, Serge Abiteboul, François Bancilhon, Nicole Bidoit, V. Delebarre, S. Gamerman, J.M. Laubin, M. Mainguenaud, T. Mostardi, P. Pauthe, et al.

To cite this version:

J. Verso, Serge Abiteboul, François Bancilhon, Nicole Bidoit, V. Delebarre, et al.. Verso: A data base machine based on non 1NF relations. [Research Report] RR-0523, INRIA. 1986. inria-00076031

HAL Id: inria-00076031
https://inria.hal.science/inria-00076031

Submitted on 24 May 2006

Rapports de Recherche

N° 523

VERSO: A DATA BASE MACHINE BASED ON NON 1NF RELATIONS

Jules VERSO

Mai 1986

VERSO : A DATABASE MACHINE BASED ON NON 1NF RELATIONS*

S. Abiteboul, F. Bancilhon[1], N. Bidoit[2], V. Delebarre,
S. Gamerman[2], J.M. Laubin, M. Mainguenaud, T. Mostardi[3],
P. Pauthe, D. Plateau[2], P. Richard, M. Scholl, and A. Verroust

Institut National de Recherche en Informatique et Automatique
78153 Le Chesnay, CEDEX
France

ABSTRACT

Verso is a database management system developed at INRIA. The main characteristics of the system are the following:

1) on-the-fly filtering is used for both unary (selection, projection), binary operations, and for updates. The filter is realized by a finite state automaton-like device.

2) data is physically organized hierarchically to take advantage of the power of the filter.

3) the user interface is based on a non-first-normal-form extension of the relational model. An algebraic language, and a screen interface are proposed.

This three aspects are closely tied. They form the basis for the originality, and the performance of the Verso database system. The version of the system discussed in the present paper has been implemented in Pascal and C under Unix on an SM90 computer. Performance studies are briefly mentioned.

RESUME

Verso est un système de gestion de bases de données développé à l'INRIA. Les caractéristiques principales du système sont les suivantes:

1) l'utilisation d'un mécanisme de filtrage par automate d'états finis, implanté près du disque. Ce filtrage permet d'exécuter efficacement non seulement les opérations unaires (sélection, projection), mais encore l'insertion, la suppression et certaines opérations binaires.

2) les données sont organisées physiquement suivant une structure hiérarchique, pour tirer profit de la puissance du mécanisme de filtrage.

3) l'interface utilisateur est fondée sur un extension non sous première forme normale (1NF) du modèle relationnel. L'usager dispose actuellement d'un langage de commandes algébriques et d'une interface plein écran plus agréable, Ever.

Ces trois points très étroitement liés font de Verso un système performant et et original. La version du système Verso présentée ici a été développée autour d'une configuration SM90 en Pascal et C sous Unix. Des études de performances sont brièvement présentées.

Present address:
1. MCC, Austin Texas, USA
2. LRI, Orsay University, Paris, France
3. CRAI, Rende, Italy

(*) This research was partially supported by the french agency ADI under contract 84077 and was part of the french joint research program on databases (PRC-BD3).

I. INTRODUCTION

The VERSO project was started at Inria. in the early eighties. with the following objectives in mind:

- Justify the approach consisting in relegating some tasks to a processor close to the mass storage device under the conventional assumption that Database Management Systems (DBMS) are I/O bound.

- Check that an automaton-like mechanism for this on-the-fly filtering capability is well adapted to query processing.

The major motivation behind such an architectural approach is to increase the performance of a relational DBMS. Although the usefulness of on-the-fly filtering has been widely accepted [CLS.OSS.S1.LSZ.ERT.Ro.Bab.IDM]. no filter has been included in a complete DBMS design to our knowledge. Our intention was therefore to develop a fully relational system that would use the above filtering concept.

To take full advantage of the filter. data is not physically stored as flat files but as a hierarchical structure called the Verso-file. This physical organization strongly suggests a logical organization of the data into non first normal form (non 1NF) relations called Verso-relations or V-relations. Indeed. several researchers have studied this concept of non 1NF relations [AB. BRS. FT. FK. Mak. SP]. It should also be noted that this notion arises naturally in the context of semantic database modelling [AH. HY]. However. the Verso system is to our knowledge the first running system based on non 1NF relations.

The query language is algebraic. All algebraic operations (except for one. namely restructuring) are performed by the filter. This filter can be viewed as a finite state automaton (FSA) which scans sequentially one or two input buffers. and writes the result of the operation on an output buffer. The restructuring operation involves some sorting. and can not be realized uniquely by the filter. The performance of the system thereby depends heavily on the performance of the filter. and on its connection to the rest of the system.

The version of the system presented here. runs under the Unix operating system. Prototypes have already been experimented on a 68000 based multiprocessor machine. the SM90. Most of the code is written in Pascal. A specialized hardware processor was first designed to realize the FSA filter. This hardware processor. connected to the mass storage as well as to the central bus is in charge of data transfer and data filtering. Later on. the hardware filter was abandoned and replaced by a standard disk exchange module including an Intel 8086 processor on which filtering is implemented by software.

Except for the use of a filter. and for the model of V-relations. the Verso system is a quite standard system:
- The data is stored in relations contained in databases. An atomic piece of data is a string of attributes of arbitrary length.
- A tree structured master index is used. Secondary indexes are not implemented.
- Concurrency is offered via the concept of transaction. and managed using two phase locking.
- Mechanisms for handling crash recovery are provided.
- Data consistency is enforced within the context of V-relation. No other constraint mechanism is provided.
- There is no security mechanism installed in the system.

The paper is organized as follows. In Section II, the Verso data model is presented. Section III deals with the architecture of the system. The user interfaces are presented in Section IV. In the last section, the performance of the (software) filter is discussed and compared to that predicted for the hardware filter.

This paper only gives an overall description of the system. The reader who is interested by description of specific aspects can refer to the following material:
- A discussion of the Verso files, and of FSA filtering [BS, BRS].
- The system architecture [B+, S].
- A formal presentation of the model together with some theoretical results on V-relations [AB, Bi].
- Performance evaluation and measures [GS, G, S, DRS].
- The description of the index mechanism [Mo], and a performance evaluation of the size of the index [Pl].
- Theoretical results on updates in [V], and the foundation of a relational interface in [Bi].
- The description of the Ever interface [Pa].

II. THE VERSO MODEL

In this section, we describe the Verso data model. We first describe the data structure called V-relation. We then present the Verso algebra. A formal presentation of the model, together with some basic results on V-relations can be found in [AB, Bi].

II.1 THE V-RELATION

In the Verso data model, the data is organized in non-1NF relations called V-relations. In a V-relation, the values of some attributes are atomic whereas the values of other attributes are V-relations of simpler structure. An example of V-relation is given in Figure II.1.

```
(COURSE ( STUDENT(GRADE)*)* (BOOK            )*)*
-------------------------------------------------
math    | toto   | A | |  | Bourbaki         |
        |        | B | |  ------------------
        |        ------- |
        | lulu   | D | |
        |        ------- |
        ------------------
comp.   | zaza            |  | Ullman         |
sci.    | mimi            |  | Delobel-Adiba  |
        ------------------   | Gardarin       |
                             ------------------
phys.   | zaza   | A | |
        |        | C | |
        |        ------- |
        ------------------
```

Figure II.1

This example describes information about courses. In each course, there are students. These students have grades for those courses. In each course, there are required books. Intuitively, this can be viewed as a relation (or table) with three columns: course, enrolment, literature requirement. The entries in the course column are atomic. The entries in the two other columns are relations of simpler structure.

Note that:

(1) There is no book required in the physics course. Thus V-relations handle null values (of this particular type at least) in a simple manner. As a consequence of this, some queries which are typically complicated to express in the relational model are simple selections in this model. An example of such a query is:
" Give all the students with no grade in CS101 "

(2) The data is naturally organized in a hierarchical manner. (It is possible to speak of the grades of a student in a course.) Furthermore this hierarchical data organization induces some implicit connection between attributes. For instance, in this example, there is a connection between books and students through course. This has some interesting consequences both at the logical and the physical levels.

At the physical level: the implicit connection corresponds to a join between what would be stored in a relational model as two relations [COURSE, STUDENT], and [COURSE, BOOK]. Thus, queries which would involve a join of these two relations would be computed just by a selection in the Verso data model thereby improving performances.

At the logical level: to ask queries like "Who are the students who have a given book", a user would typically have to specify an access path in the pure relational model (to specify which join has to be realized). This is not required in the Verso data model.

To specify the structure of a V-relation, a format is used. The concept of format is the analog of the notion of relational schema.

Formats are strings defined recursively in the following manner.

Definition: If X' is a finite string of attributes, then (X)* is a (flat) format. If X is a finite string of attributes, $f_1, \dots f_n$ are some formats for n positive, then $(X\ f_1\ f_2\ \dots\ f_n)^*$ is a format. We also require that the same attribute does not appear twice in the format.

Examples of formats:

```
(COURSE(STUDENT(GRADE)*)*(BOOK)*)*
(FILM FRENCH_TITLE(ACTOR)*(FESTIVAL(AWARD)*)*(DIRECTOR CITIZENSHIP)*)*
(WINE COUNTRY(PRODUCER ADRESS DEGREE PRICE )*)*
```

Given a format, we can define V-relations over that format, as follows:

Definition: If f is a flat format, then a V-relation over f is a set of tuples over f with atomic entries. If $f = (A_1 \dots A_n\ f_{n+1} \dots f_m)^*$, then a tuple over f is a tuple such that
- for i=1 to n, the entry is atomic, and
- for i=n+1 to m, the entry is a V-relation of format f_i.

A V-relation over f is a set of tuples over f.

Typically, a V-database will consists of several V-relations.

II.2 THE V-ALGEBRA

A simple algebra can be defined for V-relations. The reader has to keep in mind that V-relations were introduced primarily to solve some of the performance problems of the relational model. In designing the algebra, we had two requirements:
- the operations had to be mathematically sound,
- they had to correspond as much as possible to operations that could be implemented by the filter.

Indeed, we shall see that all operations but one can be computed by the filter. The unique "expensive" operation is restructuring. This operation involves some sorting. Thus the complexity of main memory computation is restricted to a unique module, namely the sorter.

The algebra consists of unary and binary operations. The unary operations are projection, selection, and restructuring. The binary ones are join, union, and difference.

We shall describe the operations mainly by examples.

Projection: Consider the V-relation of Figure II.1. Its projection over (COURSE (STUDENT)*)* is given in Figure II.2.

(COURSE	(STUDENT)*)*
math	toto
	lulu
comp. sci.	zaza
	mimi
phys.	zaza

Figure II.2

The result of a projection is required to be also a V-relation. Therefore, a format f can be projected on a set X of attributes if the string obtained by deleting from f all attributes in X, and removing empty parentheses, and the corresponding stars, is a format f'. For instance, the projection of (COURSE (STUDENT)* (BOOK)*)* on the attributes COURSE STUDENT is legal, whereas its projection on STUDENT BOOK is illegal.

Selection: The selection is more complex than the classical relational selection since it takes advantage of the richer structure. Together with classical conditions like "STUDENT= toto", it allows conditions like "exists a STUDENT", and "does not exist a STUDENT".

Selections are built using elementary conditions. An elementary condition over an attribute A is an expression of the form: A=a, A>a, A<a, A>=a, A<=a A<>a for some constant a. A condition is a boolean combination of elementary conditions.

Here are some examples of selections.

Example II.1: Consider the flat format f_1= (DATE GRADE)*. One can select grades greater than B using the condition: GRADE! > "B". One can make more complicated selections on the same format:
(GRADE! >"B" or <"D") and (DATE! > "1985/8/25")

Example II.2: Consider now the format f_2= (STUDENT(DATE GRADE)*)*. We can select the information on a particular student.
STUDENT! = "toto"
We can also use a selection of the Example II.1 to built a query on V-relations of format f_1.
STUDENT! = toto
((GRADE! >"B" or <"D") and (DATE! > "1985/8/25"))

The result will give, for the student toto, all of his grades greater than B or smaller than D obtained in an exam more recent than Aug. 25th 1985. A relation over format f_2 consists of tuples with two entries. The first entry is atomic and contains the name of a student. The second entry is a V-relation of format f_2. The selection of Example II.2 is used on that second entry.

<u>Example</u> <u>II</u>.<u>3</u>: As shown in the previous example, local selections can be used. The result of these local conditions can be forced to be empty or non empty. For instance,

```
  STUDENT!
     exist ( GRADE! > "B" or < "D" )
```

Intuitively, (GRADE! > "B" or < "D") indicates what selection to realize on the second entry of the V-relation. The term "exist" indicates that only those tuples with non empty second entry (after selection) should be kept. The result therefore contains the names of all students, and their grades (greater than B) if they got a grade greater than B.

We now explain the meaning of the exclamation marks. An exclamation mark indicates which attributes are to be projected in the result. They can be used simply as an alternative way of specifying projection.

<u>Example</u> <u>II</u>.<u>4</u>: For the format f_2, the following selection realizes the projection over STUDENT GRADE.

```
  STUDENT! ( GRADE! )
```

However, they can be used in more clever ways. For instance, consider the selection.

```
  STUDENT!
     ( GRADE! ) ( GRADE > "A" ) ( DATE < "1985/8/25" )
```

Here, GRADE is used with multiple roles: the GRADEs that are printed (which are not restricted), and the GRADEs which are compared to "A" which are used to restrict the set of STUDENTs of interest. The result of that selection gives <u>all</u> the grades of students which have a grade larger than "A", and have an exam before Aug. 25th 1985.

<u>Example</u> <u>II</u>.<u>5</u>: We finally present a more complicated example which demonstrates the possibilities of the selection. Consider the format

```
  f3= (COURSE ( STUDENT ( DATE GRADE )* )* ( PROF )* ( TA )*)*,
```

and the query:

"for each class taught by Martin, give the names and grades of all students which got an A in that class if there is no TA for that class"

In the Verso algebra, this can be done directly by a selection:

```
  COURSE!
     ( STUDENT! ( GRADE! )  exist ( GRADE > "A" ) )
     ( PROF = "Martin" )    not_exist ( TA ).
```

Now, consider the relational algebra. Suppose the data is stored in four relations CS, CSDG, CP and CTA. The query will typically involve:

- several projections (e.g., CTA on COURSE, or CSDG on COURSE, STUDENT, GRADE).
- some selections (e.g., PROF= Martin, and GRADE= A).
- a difference: to obtain the class with no TA.
- joins to combine the partial results.

The next operation deals directly with the structure of data since it allows to transform one structure into a different one.

<u>Restructuring</u>: Consider a V-relation of format film(actor)*. For each film, the actors of this film are grouped. Now, we may want to organize the data differently. For instance, we may want for each actor, the list of films where they played. This is done by the restructuring operation. In Figure II.3, an instance over film(actor)* is given, together with its restructuring according to actor(film)*.

FILM	(ACTOR)*
Purple Rose	W. Allen
	M. Farrow
Manhattan	W. Allen
Broadway	

ACTOR	(FILM)*
W. Allen	Purple Rose
	Manhattan
M. Farrow	Purple Rose

Figure II.3

Two remarks have to be made about restructuring:

- When restructuring data, some information may be lost.

- Even if loss of information is tolerated (which is typically the case of queries), some restructuring operations have no meaning. For instance, it is not possible to restructure a flat relation (A B C)* into (A(B)*(C)*)*.
A thorough study of <u>lossy</u>, and <u>lossless</u> restructuring of V-relations is presented in [AB].

We now turn to binary operations. There are essentially three binary operations: <u>join</u> (*), <u>union</u> (+), and <u>difference</u> (-). These binary operations can not be applied to arbitrary V-relations. The two V-relations must be <u>compatible</u>. Intuitively, a format can be represented by a tree. Two formats are compatible if their tree representations are subtrees of the tree representation of another format (the resulting format). For instance,
(A(B)*)* and (A(C)*)* are compatible. Also,
(A A'(B(D)*)(E E'(F)*)*(G G')*)* and (A A'(B(H)*(I)*)*(E E')*)*
are compatible: their tree representations, and a tree representation of a possible resulting format are shown in Figure II.4. On the other hand,
(A A')* and (A(B)*)* are not compatible, and
(A(B)*)* and (B(A*))* are not compatible.

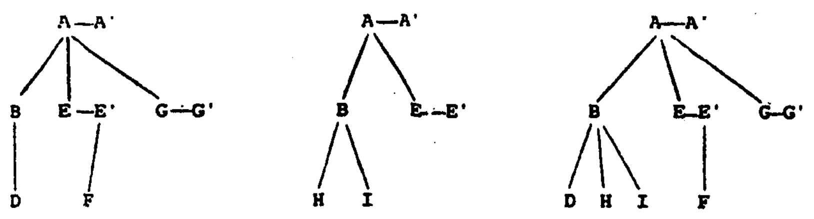


Figure II.4

Union allows to "add" the information of two instances. Join allows to "combine" the information of two instances. Finally, difference is used to withdraw the information of one instance from the information in another one. In that sense, these three operations can be seen as generalizations of the (pure) relational operations of union, join (and intersection), and difference.

Again we shall not present formally these three operations. We illustrate their effect through examples. Consider the instances I,J and K in Figure II.5(a). Some binary operations on these instances are shown in Figure II.5(b) and (c).

(FILM	(ACTOR)*)*
Purple Rose	W. Allen M. Farrow
Manhattan	W. Allen
Broadway	

Instance I

(FILM	(ACTOR)*)*
Purple Rose	W. Allen
Manhattan	D. Keaton
Broadway	W. Allen

Instance J

(FILM	(DIRECTOR)*)*
Purple Rose	W. Allen
Manhattan	W. Allen
Police	M. Pialat

Instance K

Figure II.5 (a)

(FILM	(ACTOR)*)*
Purple Rose	W. Allen M. Farrow
Manhattan	W. Allen D. Keaton
Broadway	W. Allen

I + J

(FILM	(ACTOR)*)*
Purple Rose	W. Allen
Manhattan	
Broadway	

I * J

(FILM	(ACTOR)*)*
Purple Rose	M. Farrow
Manhattan	W. Allen

I - J

Figure II.5 (b)

(FILM	(ACTOR)*	(DIRECTOR)*)*
Purple Rose	W. Allen M. Farrow	W. Allen
Manhattan	W. Allen D. Keaton	W.Allen
Brodway	W. Allen	
Police		M. Pialat

I + K

(FILM	(ACTOR)*	(DIRECTOR)*)*
Purple Rose	W. Allen M. Farrow	W. Allen
Manhattan	W. Allen D. Keaton	W.Allen

I * K

Figure II.5 (c)

III. ARCHITECTURE

We first present the hardware architecture, then we give an overall description of the VERSO DBMS.

III.1 HARDWARE ARCHITECTURE

The version of the system presented here runs on the Unix operating system and has been experimented on a 68000 based multiprocessor machine, SM90.

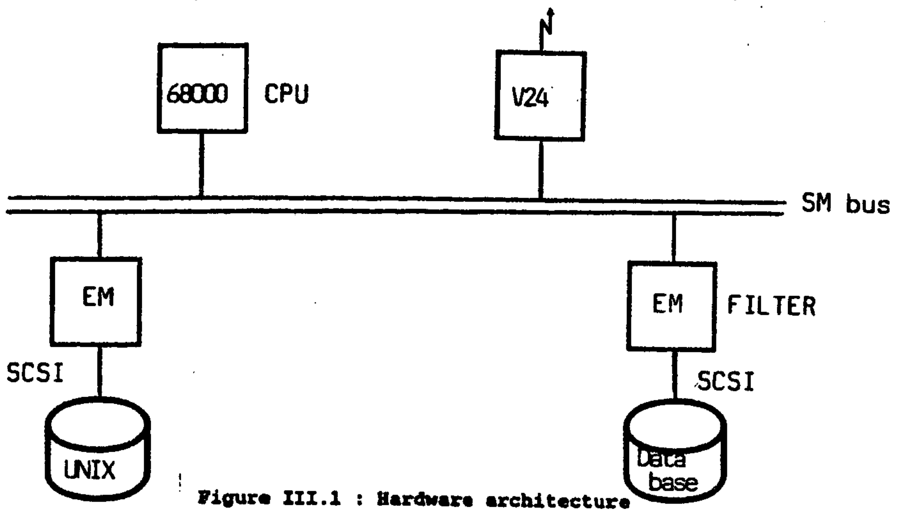


Figure III.1 : Hardware architecture

As shown in Figure III.1, the VERSO machine includes the following components, which share the central bus, the SM bus:

a) a central processing unit(CPU) including a Motorola 68000 processor, its local memory and a memory management unit;

b) a RAM memory;

c) an exchange module (EM) interfacing with a disk hosting the Unix system and the programs;

d) an user interface (V.24 or Ethernet);

e) another EM interfacing with another Disk where the databases are stored. Filtering is implemented on this EM.

The CPU is in charge of the user interface, the high level DBMS layers (to be described below) and the filter's control: it sends to the filter data transfer and filtering commands.

The filter internal structure is depicted on Figure III.2:

An Intel 8086 processor shares a local bus with an Intel 8089 processor acting as a Direct Memory

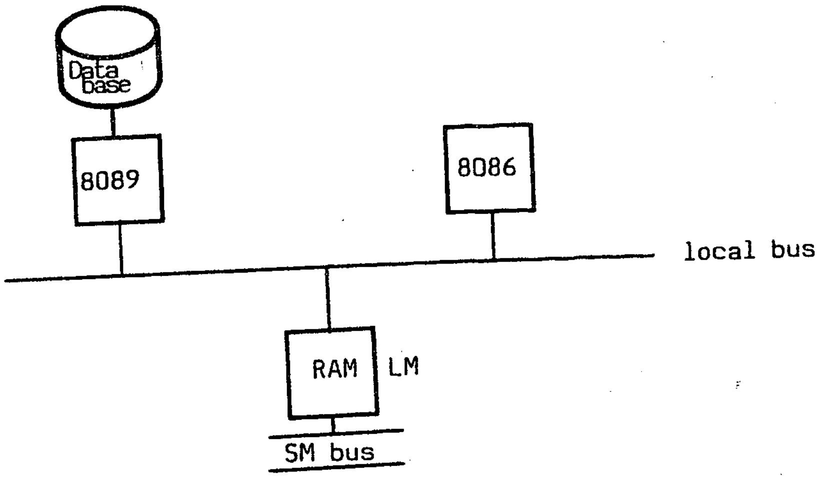


Figure III.2 : Filter architecture

Access (DMA) between the disk controller and an 128 Kbytes RAM local memory (LM). Memory LM is shared among the Intel processors (local access) and the CPU (global access through the SM bus). Memory LM hosts buffers (data to be filtered and filtering results) as well as filtering code swapped in from the disk at the beginning of a session. The 8086 processor is in charge of filtering and data transfers between the disk and memory LM.

The CPU accesses memory LM for i) sending commands to the filter, ii) reading filtering results and iii) inserting data into the database.

The last component on the local bus is an EPROM memory in which a real time system and low level tasks are stored.The main "system" tasks are the interaction with the CPU and the disk input/output tasks. Filtering is implemented by means of 6 tasks written in Pascal and developed from an HP64000 development system.

III.2 SOFTWARE DESCRIPTION

In terms of functionality, Verso is a fairly standard system: it offers data definition, search and manipulation, transaction management, concurrency control and recovery and simultaneous access from separate sites .

The three latter functions will only be roughly sketched, since classical solutions have been chosen for those problems. The interested reader is referred to [B+] for more details.

As usual, a transaction is a sequence of requests (Verso commands, see Section 4). The system accepts interleaving of requests issued from different transactions, but requests are sequentially run. In order to improve the global throughput, pipelining of requests on a single CPU is under study.

A regular two-phase locking protocol is used, together with deadlock prevention. Physical locking has been chosen with granularity of one block (one disk track). However the index is locked only for the duration of the index request (and not until the end of transaction, as for a regular data access). The concurrency control algorithm is based on the shadow system[L]: each transaction processes its work in a shadow area; updating is done through an address translation mechanism and does not necessitate any actual rewriting on the disk.

We will describe in more details the data search and manipulation functions.

The Verso system consists of three layers:

1) The highest level is the V-relational level (see Section II): the objects seen at that level are the V-relations and the schema. There are four types of V-relations: input, output, temporary and base V-relations. The schema contains the description (format) of all V- relations. The operations at that level were described in Section II.

2) The second level is the file level: the objects defined at that level are Verso files or physical representations of V-relations and the non-dense Index which permits to locate data. A file is characterized by a name and a list of block addresses. All files are completely sorted in lexicographic order. Files are then partitionned into blocks. Each block corresponds to a disk track (8 or 16 Kbytes), which is the smallest addressable unit. The index structure will be described in Section III.2.2. The operations at that level are:
   i) index manipulation in order to locate a V-relation,
   ii) selection/projection, insertion, deletion into/from a file (corresponding to a unary operation on V-relations);
   iii) binary operations on files (corresponding to a binary operation on V-relations);
   iv) file sort (corresponding to restructuring, see section 2). We use a merge-sorting algorithm: once each block has been sorted, blocks are merged. This merge is a file union performed in linear time by the filter.

3) At the lowest level, we find a block characterized by its address. There are two kinds of operations at the block level: filtering (see section III.2.3) and internal sort of a block. As mentionned earlier, this operation of complexity NlogN is not performed by the filter.

III.2.1 Query processing

Let us take the example of the V-selection to illustrate query processing through the three layers as well as the splitting of tasks between the CPU and the filter.

A V-selection is submitted to the system. At the first level, given the V-relation name ,the schema is searched to get the V-relation format. Two operations are then performed:

i) compile the query into an FSA to be loaded into the filter memory (LM);

ii) search the index in order to get a subset of the blocks of the V-relation that have to be filtered. The result of this index search is a list of one or more block addresses.

The above processing is performed by the CPU. Once the FSA corresponding to the query has been loaded into memory LM, the CPU initiates filtering: it sends to the filter together with a selection command a list of block addresses if the source file corresponds to a temporary or base relation. Furthermore, the CPU sends a list of target block addresses where to store the filtered data if the target file corresponds to a target relation of type base or temporary. In the case where the source (target) relation is of type input (output) the blocks are written (read) one after the other into (from) memory LM upon the filter demand.

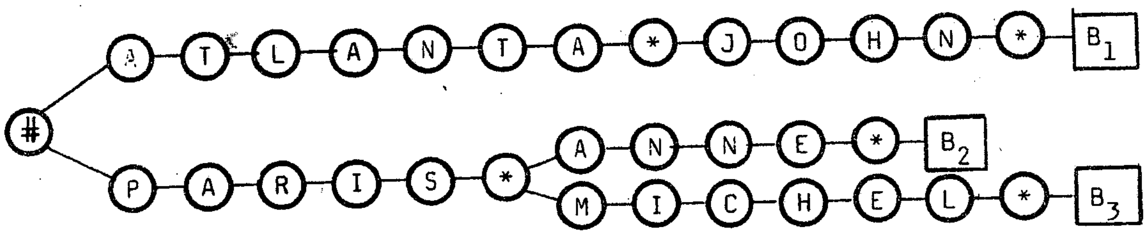
To summarize, V-relation operations are performed by the CPU, including transaction management and concurrency control. The CPU is also in charge of Index operations, as well as FSA generation and loading.

The filter is in charge of file and block level operations on data (except internal sort of a block). Binary operations can also be performed in linear time since the files are sorted: indeed, if both source relations have "compatible" formats (see Section II.2), there exists a FSA like mechanism which allows to scan both source relations one byte at a time in order to perform union or join or difference.

III.2.2 Index structure

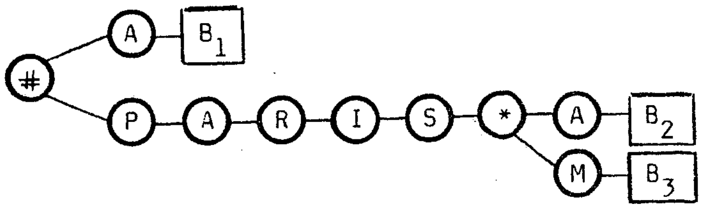
Recall that all V-relations are sorted and partitioned into blocks. The master index is a non dense index which stores for each block, the whole smallest tuple of the block (and not its key) followed by the block address. However the stored information is compacted through a trie structure we now describe. The resulting index is small enough to be maintained in RAM memory. It is shown in [P1] that under reasonable assumptions in the case of a 300 Megabytes file, the average index is approximatively 100 Kbytes.

In order to describe the index structure, we take as an example the flat V-relation (TOWN, NAME) with two attributes, and assume it is stored as a file of size 3 blocks, such that the first tuples of each block are respectively ATLANTA, JOHN; PARIS, ANNE and PARIS, MICHEL. To this set of tuples corresponds the following trie :



We assume that the attribute values are bounded by a special character "*" and all tuples are preceded by "#". To each leaf of this trie, we append a block address.

To keep the amount of information minimal to distinguish two blocks we prune the linear subtrees of the TRIE (except the block address) :



This tree may be linearly represented as follows :
#(A & B1 PARIS * (A & B2 M & B3))
where each time a node has more than one child we add parentheses and the special character "&" indicates a leaf node.

The index has been first implemented by means of the above linear representation of the pruned trie in order to be able to perform each index operation using the FSA filter. A FSA was exhibited for each (search and update) index operation. However update operations turned out to be rather complex and cumbersome. A tree representation of the pruned trie structure - although a bit less compacted - was then preferred [Mo]. Index operations are implemented by classical tree operations.

III.2.3 Filtering

Recall that the filter sequentially scans a source buffer and writes into a target buffer the relevant data. In the case of insertion or binary operations, two source buffers are concurrently scanned. Three processes are pipelined: the operations of (i) loading the sequence of source blocks, (ii) filtering these blocks and (iii) unloading the resulting blocks either onto disk or to the user.

There exists two pools of buffers. The loader sequentially loads source blocks into buffers requested from the source pool. Once a block has been loaded, filtering may proceed on that block concurrently to loading of the following block. Once a target buffer is full, the unloader can unload this buffer while filtering resumes on another target buffer requested from the target pool.

The FSA filtering principle has been thoroughly described in [BS]. It was shown in [BS,S] that an automatonlike device is sufficient to perform on the fly the V-algebra operations. In the case of binary operations, the V-relation formats must be compatible. We give below a sketchy description of the filtering principle for the selection/projection operation.

Filtering is based on data recognition by a FSA. Given a request for an algebraic operation and the V-relation format, a compiler generates an automaton to be loaded into the filter's memory LM.

Then the filter scans the V-relation in the source buffer one byte at a time. Before reading a character, the automaton is in a given state. To the (state, character) couple corresponds a next state and an output function. Roughly speaking, the latter consists of the updating of the register BAR where to write the next character.

In the case of hardware filtering [B+], a cycle of the filter starts when, in a given state, a character is read. The filter's memory is a 256-line (256 states) by 256-column (256 bytes) matrix. The (current state, current character) couple addresses a word in the matrix which contains the next state plus the output function to be interpreted and run before going to the next state.

In the case of software filtering [S], the source buffer is not anymore scanned one byte at a time: during a cycle, an entire attribute value is scanned. There are three types of cycles: (i) if the corresponding attribute value has neither to be projected nor to be compared to one or several values, then the attribute value is just skipped; (ii)the attribute value is compared to a set of values and it is possibly written into the Target Buffer; (iii) the attribute value is just copied from Source Buffer to

Target Buffer.

In the case of a binary operation, two source buffers are concurrently scanned. The filter analyzes a character read in one of the two buffers and possibly writes one character into the Target Buffer.

IV. THE USER INTERFACES

In this section, we present various user interfaces available in the Verso system. We first describe the command language VERSO. This language is used as the unique language of communication between the system, and the rest of the world. Indeed, other interfaces can be viewed as translation modules between more user friendly interfaces and the VERSO language. We then present the full screen interface which is oriented toward non sophisticated users, or simply users which want to use the system, and not be used by the system. Finally, we briefly describe a Pascal extension which allows calls to the database system.

IV.1 THE COMMAND LANGUAGE

We just give a brief description of the command language. A complete description can be found in [VERSO].

A command starts with an integer which identifies the user (site) issuing the command. (The site number is generated by the system.) This integer is followed by an integer identifying the transaction number if the command is given from a particular transaction. For instance,

```
> 12 3 validate;  (* transaction 3 in site 12 is validated *)
> 12 stop;        (* the site 12 is leaving the db system  *)
```

Most commands must be given from within a transaction. A transaction is a sequence of commands issued from a given site with the same transaction number starting by a "start" command, and ending with a "validate" or "abort" command. The command "return" allows to abort only part of a transaction if abort steps were previously used. We now illustrate these concepts by an example of transaction.

```
> 12 3 start;
> .....                                (* part1 *)
> 12 3 step;
> .....                                (* part2 *)
> 12 3 step;
> .....                                (* part3 *)
> 12 3 return 2; (* return two steps backwards: part2 and part3 are aborted *)
> 12 3 validate;                (* only part1 is validated *)
```

In the command language, a V-relation is specified by its name, and the database to which it belongs. For instance,

```
cinema.new_film    (* V-relation film in base cinema     *)
temp.xsdwe         (* V-relation xsdwe in base temp      *)
```

The data definition language allows the creation of a new base,the creation of a new V-relation (specifying its format), the modification of the format of some existing V-relation, or consult the schema. One can modify the structure of an existing V-relation by renaming some attributes, or adding some new attributes. Examples of these various commands are now given. (For the sake of simplicity, the site and transactions numbers are omitted when not necessary to the presentation.)

```
> create_base cinema;
> create cinema.film (film (acteur)*(directeur)*)*;
> rename cinema.film acteur:actor, directeur:director;
```

```
  > consult cinema.*;
  verso: base cinema
         film : (film (actor)*(director)*)*
  > redefine cinema.film (film (actor)*(director)*(producer)*)*;
  > consult cinema.*;
  verso: base cinema
         film : (film (actor)*(director)*(producer)*)*
```

The processing is done by a unique command which assigns to a target relation the result of some algebraic operation on one or two source relations. The operations are essentially the operations described in Section II.

```
   > consult cinema.*;
   verso: base cinema
          director : film(director)*
          actor    : film(actor)*
          films    : director(film)*
 (* projection    *)
   > assign s.r1 := cinema.actor ( film );
 (* restructuring *)
   > assign cinema.film := restructure cinema.actor;
 (* selection     *)
   > assign s.r2 := cinema.actor such that
             film! exist_pas ( actor="D. Hoffman" );
 (* join          *)
   > assign s.r3 := cinema.actor * cinema.director;
 (* union         *)
   > assign s.r4 := cinema.actor + cinema.director;
 (* difference    *)
   > assign s.r5 := cinema.actor - cinema.director;
```

Insertion and deletion are offered for database updates. One may insert a tuple, and delete one or more tuples according to some complex conditions. We now give examples of insertions and deletions :

```
   > insert cinema.actor film="Kramer vs Kramer", actor="D. Hoffman";
   > delete cinema.films such that director="Truffaut" or ="Godart";
```

Some other commands are of course available which are neither in the DDL nor in the DML. These commands allow among other things to:
- communicate with other users of the database system.
- enter some debug modes.
- call an on-line manual.
- escape to a Unix shell.
- exit the system.

IV.2 THE SCREEN INTERFACE EVER

In this section we present the EVER interface. A complete description can be found in [Pa].

EVER is a multi window screen interface tailored to answer the various needs of a dialogue with the Verso system. In particular, four modes are offered:

- a mode for command edition,
- a mode for selection/projection edition,
- a mode for data edition, and
- a mode for format edition.

A transaction is associated to each window. (In particular, the user can work on two separate transactions in separate windows). In the command mode, EVER gives the user more flexibility by offering the full power of a text editor. Perhaps the most useful aspect of this mode is its guidance of the user in the tedious task of specifying the commands. When necessary, the system switches automatically to some other modes. For instance, if the user indicates that he/she wants to create a new relation, the system goes into the format edition mode.

In the format mode, a format can be very easily specified. The normal moves in text edition (forward, backward, upward, and downward) are replaced by moves in the format (i.e. a tree) : move to the parent, to the first child, to previous or next brother.

The data mode allows to have direct access to the data in a V-relation. It can be used for browsing through data, or for updates. The major difficulty of browsing in this context is that there is no restriction in Verso neither on the length of atomic values, nor on the number of attributes, nor on the depth of the format. It is typically the case that the screen is not large enough to represent the data. Two choices are thus made:

length of atomic values: On the screen, a buffer of finite length is allocated for each atomic entry of a given attribute. If the value does not fit in that buffer, only part of it is represented on the screen. The user can access the remaining information by scrolling in the buffer.

complexity of the format: if EVER can not fit everything in a window, it chooses to represent some projection of the V-relation of interest. The user can of course override this choice, and prefer a different projection from the one chosen by EVER.

In the data mode, the user can use search (forward and backward) for a given value. Insertion and deletion can be performed directly on the data. In the case of insertion, the inserted data is kept by EVER, and sent to Verso when the user "writes" the V-relation. In fact, except for the particular nature of the data, the editing of a V-relation ressembles the editing of a text in a conventional editor like Vi or Emacs.

A last mode, the selection/projection mode, is entered when the user wants to perform one such operation on a V-relation. The projection is defined first. The specification of the selection is then similar to the input of data in a V-relation. The only difference is that conditions are entered instead of values. In that respect, the specification of a selection in Verso is done like in query-by-example [Z].

IV.3 THE V-PASCAL INTERFACE

In this section, we present an extension of Pascal, V-Pascal [Mai], which combines the advantages of the Verso system, and that of the Pascal programming language [PASCAL].

The V-Pascal interface uses the notion of relational skeleton. Given a Verso format f, the relational skeleton R is the relational schema which can represent the same information. For instance, consider the format f= (film(actor)*(director)*)*. Its corresponding skeleton is the relational schema R= (film, film actor, film director).

Programs in V-Pascal are compiled into conventional Pascal programs with calls to a Verso library. Since Pascal does not allow the direct use of complex data structures like V-relations, the treatment is done one flat tuple at a time. In fact, the Pascal program can work with the relations of the skeleton. These relations are viewed as ordered. The Pascal program has access to the information contained in these relations using some functions (get_first, get_next, get_previous,...).

The V-Pascal interface has been tested on a particular application. Some agregate functions not provided by the Verso system were needed in the application. V-Pascal could realize them quite easily. V-Pascal has also been used to evaluate some alternative join algorithms.

A pure relational interface is under study. Like V-Pascal, it will use extensively the notion of format skeleton. The theoretical foundations of such an interface are exhibited in [Bi]. It is shown there that an arbitrary relational query q to R can be translated into a Verso query q' on f. Some optimization on the resulting Verso query based on a tableau technique is also proposed in [Bi].

V. PERFORMANCE

Two prototypes of filters were realized for the VERSO system. In the first prototype, the data access function was implemented by means of dedicated hardware. In the current prototype, filtering is implemented by software on an Intel 8086 processor. The performance evaluation work presented in this section focusses on the problem of choosing between these two competitive approaches for implementing a performant relational DBMS.

At the time where the performance study was started no real life measures were available ; modelling was used for evaluating the filter's response time to a query in both architectures. Such studies were useful to help the designers in their choice of the best parameters within each architecture. These studies are reported in [G,GS,S]. However when comparing the two architectures, the software filter turned out to provide an acceptable performance although inferior to that of hardware filtering.The latter approach then appeared as deceiving compared to its design complexity. One of the conclusions of the study reported in [G] was that the hardware filter's power is badly utilized.

Later on, when the VERSO system was operational with a software filter, two types of measures were performed.

a) 10 relational queries were run on a real life database provided by the french ADI agency. The database volume was of the order of 1.5 Mbytes. The database was stored on a 5Mbytes disk. The only measure performed was the query response time.

b) The filter was then tested against a benchmark adapted from that designed at the University of Wisconsin [Bit] and whose objective was to allow the comparison between several research prototypes and industrial products.

Currently only the selection/projection operation has been thoroughly evaluated with this benchmark.

The main conclusions on software filtering design drawn from this measurement experiments are presented in Section V.1 and compared to the predictions reported in [GS]. Measures of the hardware filter are not yet available; a comparison between the (measured) software filter's response time and the (predicted) hardware filter's response time is attempted.

A comparison between the Verso system and other DBMS is presented in Section V.2. As a basis for the comparison of the response times,

- we restricted ourselves to the selection/projection operation, and

- we use the figures reported in [Bit].

Measurement on other operations (e.g. join) as well as a comparison with other DBMS is under study.

Section V.3 addresses the issue of the performance improvement provided by relegating filtering tasks to a separate processor.

V.1 FILTER'S RESPONSE TIME

The response time was measured as the time elapsing between a CPU command (once the FSA has been loaded) and the "end of filtering" acknowledgment by the exchange module. Using the same benchmark as in [Bit], a 10 000 tuples relation was taken as a source relation, where each tuple is 182 bytes long (the relation of size 1.82 Mbytes is stored on 185 blocks). The Source Relation has 14 attributes. The measurement reported in [DRS] thoroughly studies the influence of the number of attributes projected, the number of attributes included in the selection criteria, etc. We only present here the case where all attributes are projected and the query is : $c_1 < x$ where c_1 is the first attribute and x ranges between 1 and 10 000 (c_1 is a key). We restrict our attention to the case where the target relation is an output relation and therefore is read by the CPU. Since c_1 is the first attribute, only a subset of the 185 source blocks were selected by the index and were scanned by the filter (1 if x = 10, 185 if x = 10 000).

V.1.1 Response time

The filter's response time is plotted versus x in Figure V.1 .

The response time is linear in x, i.e. in the number of bytes scanned. If the query is very selective ($x < 100$, i.e. less than 1 per cent of the tuples are selected), the response time is of the order of 1 second. On the contrary, scanning the whole relation would require more than 7 minutes.

V.1.2 Cycle time

Recall the filter sequentially scans the source data. While the hardware filter scans the source data one byte at a time, this is not true anymore for the software filter. Indeed the time spent on an attribute value depends on whether this attribute is to be projected or compared to a given value or just skipped.

Define the cycle time to be the time spent by the filter scanning one byte. With the hardware filter, this time denoted by τ_M may be considered as a constant independent of the query and predicted to be equal to 500γs. On the contrary with software filtering the cycle time has a significant variance. By extension, we call in the latter case cycle time, the ratio of the response time over the number of scanned bytes. For the above query, the cycle time denoted by τ_s is equal to 300 μs.

In [GS], τ_s was predicted to be of the order of 3 to 5 μ_s ! The descrepancy between the measure of τ_s and the rather optimistic prediction in [GS] may be partly explained as follows : the current filter's prototype has been written in a poor Pascal and runs on a 5MHz Intel 8086 microprocessor. By using a faster microprocessor and writing the code in a more performant Pascal (with critical sections written in assembly language), one should increase the performance by one order of magnitude : $\tau_s \simeq 30\,\mu$s.

V.1.3 Hardware versus Software Filtering

With the software filter, filtering a block takes a significantly longer time than reading it from (writing it onto) disk : the processes of loading, unloading a block are idle most of the time : I/O time is "hidden" behind filtering time itself and the response time is approximately the filtering time itself.

With hardware filtering, on the contrary, the filter itself is idle part of the time waiting for a block to be read from (written onto) disk. The response time then is approximately the time to transfer data between disk and memory [GS].

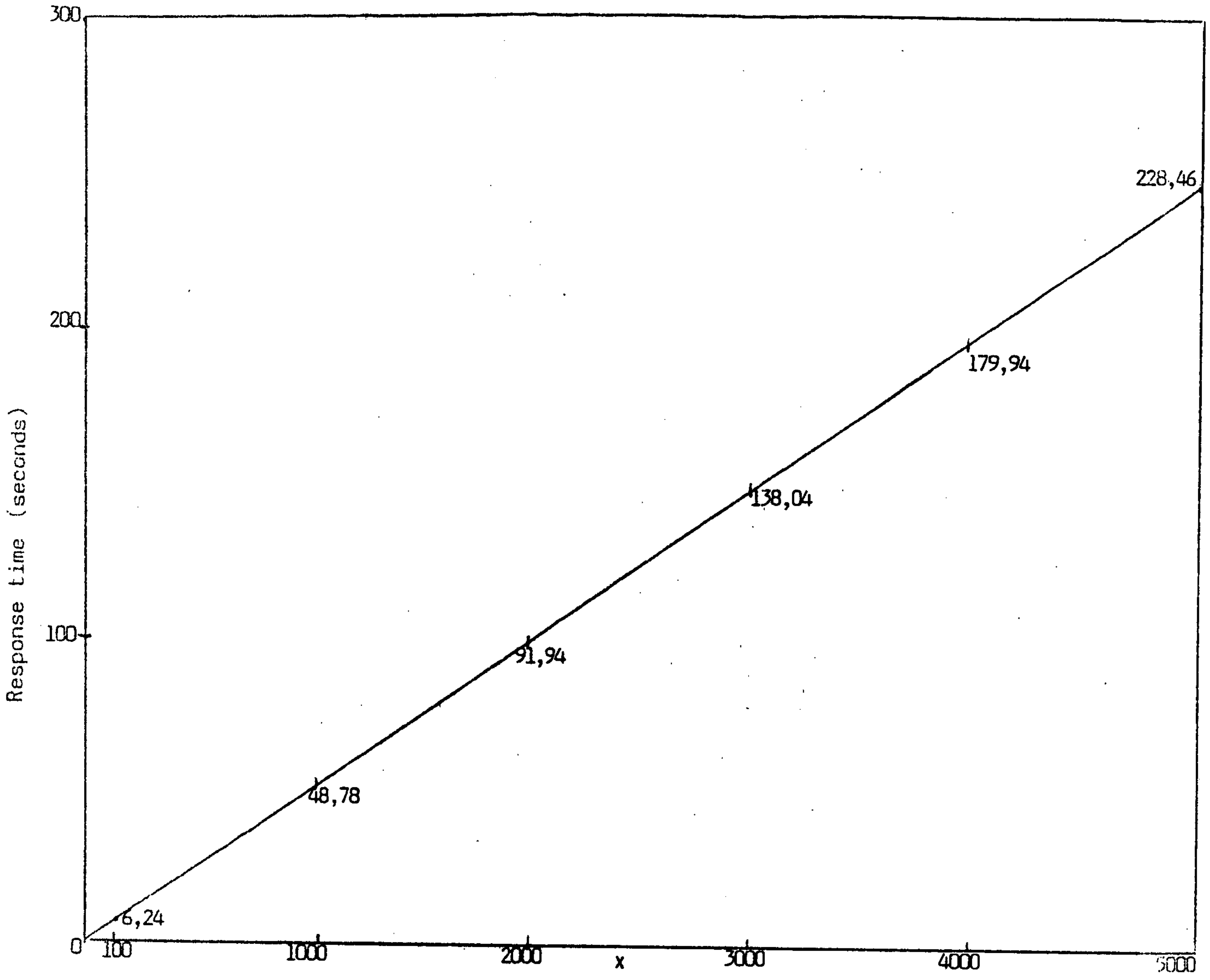


Figure V.1

Denote by τ_D the time to transfer a byte from disk to memory : with the current SMD disk interface connected to the hardware filter (10 Mbytes/s) τ_D = 800 γs (see [G .GS] for a detailed study of the hardware filter).

Then a reasonable estimate of the ratio of software filter's response time over hardware filter's response time is the ratio R = $\tau s/\tau_D$. For the above selection query, R $\simeq$ 37 !

Recall however, no measures are yet available for the hardware filter and the above value of τ_D is probably rather optimistic.

In summary, the VERSO hardware filter should be extremely faster (more than 20 times faster) than an optimized version of the current filter implemented by means of an "off-the-shelf" processor. This is in contradiction with the modelling study reported in [GS]. In this study the predicted performance of the software filter was rather optimistic.

V.2 COMPARISON WITH OTHER DBMS

For this experiment, the global response time to a selection query initiated by the user at the terminal was measured with the Verso system and compared to the response time provided for the same query by the following systems: the commercial version of the INGRES DBMS [St], the relational DBMS ORACLE (Version 3.1) , the research database machine DIRECT [De] and the IDM500 database machine [IDM].

We followed the experiment reported in [Bit], using the same database and selection queries for the Verso system. The response time figures for the above 4 DBMS were taken from [Bit, Tables 1, 2, and 3]. We give below the main characteristics of the experiment. For more details, the reader is reported to the above reference.

The four systems were implemented on Vax 11/750 computers except for the IDM500 machine which was connected to a PDP 11/70 host. Observe that those computers have more powerful processing units than the SM90 machine on which the Verso system is implemented. However precise evaluation was not available for comparing the power of a Vax 11/750 CPU to the Intel 8086 processor.

The selection queries were run on a 10,000 tuple relation where each tuple is 182 byte long. The response time (in seconds) presented in Tables V.1, V.2 and V.3 represent an average time based on a test set of ten different queries with three selectivity factors: .01%, 1% or 10% of the source tuples were selected. The relation is sorted on the first attribute is a key and a non dense index is constructed on this key.

Note that the relation is flat and then is not adapted to evaluate the advantages of the Verso data organization. In particular the compaction of data may significantly decrease the response time.

Table V.1 presents the response time when in the selection criteria there is neither c_1 nor any attribute corresponding to a dense index. On the contrary, Table V.2 gives the response time to a query including c_1: a search through the non dense index restricts the number of tuples to be evaluated. Table V.3 reports a variant of the experiment of Table V.1 where the result tuples are to be displayed on the user's screen (the display time is omitted).

One can draw a number of conclusions from the results presented in these tables. Without index (Table V.1) the response time provided by the Verso system is independent of the selectivity factor since the entire relation is sequentially scanned whatever the selectivity factor is. Compared to the other systems, Verso's response time is rather large: the main reason for this is that the filter's 8086 processor is very slow.
On the contrary, when using a non dense index (Tables V.2 and V.3), despite the mediocrity of the 8086 processor, the Verso system provides an acceptable response time although superior to that of INgres of IDM500. the extracost incurred in displaying the results is very small.

V.3 PARALLELISM BETWEEN CPU AND FILTER

If both Source and Target Relations are stored on disk, once the filtering command has been delivered to the Filter, the CPU is idle for other tasks.

System \ Selectivity	1 %	10 %
C -INGRES	38.4	53.9
ORACLE	194.2	230.6
IDM	21.6	23.6
DIRECT	43	46
VERSO	192	192

Table V.1 : Selection without index

System \ Selectivity	1 %	10 %
C -INGRES	3.9	18.9
ORACLE	16.3	130
IDM	1.5	3.7
DIRECT	43	46
VERSO	6	26

Table V.2 : Selection with index

With the current architecture of the VERSO system, requests are sequentially run. Then while filtering is under process, the CPU is totally idle, unless the machine is shared by other non-DBMS users.

Relegating filtering to a separate processor has however several interests :

1) On-the-fly filtering: Filtering is much faster when implemented on a dedicated unit, if the latter is faster than the CPU. This is certainly true when dedicated hardware is used (see above). However, no measures are yet available for the Verso system, in order to prove filtering would be much slower if implemented on the CPU.

2) Multifilters: Another simple extension to the current architecture would be to use several "off-the-shelf" filters for answering the same query. Such a Single Instruction Multiple Data (SIMD) approach often proposed [De,LSZ,LK] assumes data are split on several disk units, each disk unit being interfaced to a filter. With such an approach, the decrease in response time should depend on the query and on the disk allocation strategy.

Selectivity / System	0.01 % (1 tuple)	1 %
C - INGRES	0.9	5
ORACLE	2.5	27
IDM	0.7	2.7
DIRECT	46	49
VERSO	2.3	6.5

Table V.3 : Selection with index. display on the screen

3) <u>Multiinstructions</u>: The above approaches increase the global throughput (measured in number of requests per unit of time) by decreasing the query response time. There exists a complementary approach, namely that of accepting into the system several requests at a time. This can be done by pipelining the various stages of a request execution on a single CPU. The simplest way of pipelining requests, is that, when the filter processes a request, the CPU processes another one. We currently study such an improvement of the VERSO system. Another more ambitious approach for running several requests at a time is to utilize several CPU's (say one for each request). This approach has been followed in some research prototypes [BHK,S+,Gamma] and announced in recent commercial products such as Teradata's database computer [Sh] or japanese and french products [AC, K+].

VI SUMMARY

This paper was devoted to the presentation of the Verso database machine. prototypes of which run under the Unix operating system on a 68000 based machine. the SM90. With respect to more classical relational DBMS designs. VERSO major novel features are the following:

1) It includes a filter implemented on a separate processor close to the mass storage device. This filter is in charge of all algebraic operations except for restructuring. This automaton-like mechanism is extremely well adapted to processing of both unary. and binary operations. Furthermore. the filter is also used for providing fast updates.

2) Data is organized in non 1NF relations. This allows to combine the advantages of the relational model (e.g., an algebraic language). and the possibility of hierarchical data organization. To our knowledge. the Verso system is the first running system based on non 1NF relations.

The first objective of the VERSO project was to justify the approach consisting in relagating filtering to a fast processor close to the mass storage device. For that purpose. a specialized hardware processor was first designed to realize the filter. For cost and portability reasons. this processor was then abandoned and replaced by an Intel 8086 microprocessor on which filtering was implemented.

The first response time measurements clearly show that the VERSO system is not faster than commercial systems such as ORACLE or INGRES. The main reason is that the 8086 microprocessor on which filtering was implemented is slow. By using dedicated hardware for filtering. one should gain at least one order of magnitude on response time. However. standard microcomputers have a performance that increases rapidly with time. For that reason. following [BD]. we believe that the use of "off-the-shelf" components for filtering should be preferred to a time-consuming and costly design of dedicated hardware.

Besides. this first experience with a non 1NF model is quite promising. Verso users seem to adjust quite fast to those more complex structures. For instance. it turned out that although the Verso language was not intended to be user friendly. it didn't require too much practice to be capable of writing even complex queries in that language. Not surprisingly. the screen interface EVER has been quite an improvement for users.

REFERENCES

[AB] Abiteboul S., Bidoit N., : "Non First Normal Form Relations to Represent Hierarchically Organized Data", Proc. of ACM-SIGMOD Conf. on Principles of Database Systems, Atlanta, 1984, pp. 191-200 (To appear in Journal of Computer Science and Systems).

[AH] Abiteboul, S., R. Hull, : "IFO: a Formal Semantic Database Model", Proc. of ACM-SIGMOD Conf. on Principles of Database Systems, Waterloo, 1984.

[AC] Armisen J.P., Caleca J.Y., : "A commercial Back-End Data Base System", Proc. Inter. Conf. on Very Large Data Bases, Cannes, 1981.

[B+] Bancilhon F. et al : "VERSO : A Relational Back End Data Base Machine, San Diego, Sept. 1982 ; also in Advanced Database Machine Architecture, D.K. Hsiao editor, Prentice-Hall 1983, pp.1-18.

[Bab] Babb E., : "Implementing a Relational Database by Means of Specialized Hardware", ACM Trans. on Database Syst., Vol. 6, no 2, 1981.

[BD] Boral H.,DeWitt D.J.,: "Database Machines: An Idea whose Time has passed. A Critique of the future of Database Machines"' in Database Machines,H.O. Leilic and M. Missikoff editors,Springer-Verlag,1983 pp 166-187.

[BHK] Berra P.B., Oliver E., : "The Role of Associative Array Processors in Database Machine Architecture", Computers, Vol. 12, no 3, 1979, pp.53-61.

[Bi] Bidoit N., : "Un Modele de Donnees Relationnel Non Normalise : Algebre et Interpretation", These 3e cycle, Universite Paris-Sud, 1984.

[Bit] Bitton D. et Al., : "Benchmarking Database Systems : A Systematic Approach", Computer Science Department, Technical Report no 526, University of Wisconsin, December 1983.

[BRS] Bancilhon F., Richard P., Scholl M., : "On Line Processing of Compacted Relations", Proc. Inter. Conf. on Very Large Data Bases, Mexico, 1982.

[BS] Bancilhon F., Scholl M., : "Design of a Backend Processor for a Database Machine", Proc. ACM-SIGMOD, Santa Monica, May 14-16, 1980, pp. 93-93g.

[CLS] Copeland G.P., Lipovski G.J., Su S.Y., : "The Architecture of CASSM : A Cellular system for Non-numeric Processing", Proc. 1st Annual Symposium on Computer Architecture, Dec., 1973, pp. 121-128.

[De] Dewitt D.J., : "DIRECT - A Multiprocessor Organization for Supporting Relational Database Management Systems", IEEE Trans. on Computers, vol. C.28 no 6, June 1979, pp. 395-405.

[DRS] Delebarre V., Richard P., Scholl M., : "Filtrage des Donnees dans les SGBD Relationnels : L'Experience VERSO", In Nouvelles Perspectives de Bases de Donnees, Eyrolles, 1986.

[ERT] El Masri A., Rohmer J. et Tusera D., : "A Machine for Information Retrieval", Proc. 4th Workshop on Comp. Arch. for Non-numerical Processing, Syracuse, New York, August 1978.

[FT] Fisher, P., S. Thomas, : "Operators for Non-First-Normal-Form Relations", Proc. of the 7th International Comp. Soft. Applications Conf., Chicago, 1983.

[FK] Furtado, A., L. Kerschberg, : "An Algebra of Quotient Relations", Proc. of SIGMOD, Toronto, 1977.

[G] Gamerman S., : "Ou l'on decouvre que les performances des Filtres dans les Machines Bases de Donnees ne sont pas celles que l'on croyait". These de 3e cycle, Universite de Paris-Sud, Juin 1984.

[GAMMA] Dewitt D., Gerber R., Graefe G., Heytens M., Kumar K., Muralikrishna M.: "Gamma: a High Performance Dataflow Database Machine ", Internal report, CSD, University of Wisconsin, 1986.

[GS] Gamerman S., Scholl M., : "Hardware versus Software Data Filtering : The Verso Experience", Proceedings of the Fourth International Workshop on Database Machines, Grand Bahama Island, March 6-8, 1985, pp. 112-136.

[HY] Hull, R., C. Yap, "The Format Model: A Theory of Database Organization", Journal of the Assoc. for Comp. Machinary, 1984.

[IDM] IDM 500 Reference Manual, Britton-Lee Inc., Los Gatos, California.

[K+] Kakuta T. et al. : "The design and Implementation of the Relational Database Machine Delta", Proc. of the 4th International Workshop on Database Machines, Grand Bahamas Island, March 6-8, 1985, pp. 13-34.

[L] Lorie R.A. : "Physical Integrity in a Large Segmented Database", ACM-TODS , Vol 2, no 1, March 1977.

[LK] Lecalve A., Kalfon P., : "S.A.R.I. : Systeme Associatif de Recherche d'Informations", Actes des Journees Machines Bases de Donnees, Sophia-Antipolis, Septembre 10-12, 1980, pp. 85-104.

[LSZ] Leilich H.O., Stiege G., Zeidler H.Ch., : "A Search Processor for Database Management Systems", Proc. Inter. Conf. on Very Large Data Bases, 1978, pp. 280-287.

[Mac] MacLeod, I.A., A Database Management System for Document Retrieval Applications, Information Systems, 6.2, 1981.

[Mai] Mainguenaud, M., : "Immersion de Primitives d'accés à un SGBD dans un langage de haut niveau", Mémoire d'ingénieur IIE ,juin 1985.

[Mak] Makinouchi, A., "A Consideration on Normal Form of Not-Necessarily- Normalized Relations in the Relational Database Model", : Proc. Inter. Conf. on Very Large Data Bases, Tokyo, 1977.

[Mar] Marayanski F., : "Backend Database Systems", Computing Surveys, Vol. 12, no 1, 1980.

[Mo] Mostardi, T., : "Un Indice Compatto per una Macchina per Basi di Dati Relazionali Progetto ed Implementazione", Proc of annual conf AICA, Florence, Oct 1985.

[OSS] Ozkarahan E.A., Schuster S.A., Smith K.C., : "RAP - An Associative Processor for Data Base Management", INPG, Grenoble, 1979.

[PASCAL] Wirth ,N., : " The Programming Language Pascal", Acta Informatica,3 ,1971, pp 35-63.

[Pa] Pauthe P., : "EVER, un editeur de V-Relations", These de 3e cycle, Universite Paris-Sud, 1985.

[Pl] Plateau D., : "Une structure compacte pour indexer un fichier totalement ordonne : evaluation et mise en oeuvre", These de 3e cycle, Universite de Paris-Sud, 1983.

[Ro] Robert D.C., : "A Specialized Computer Architecture for Text Retrivial", Proc. 4th Workshop on Comp. Arch. for Non-numerical Processing, Syracuse, New York, August 1978.

[SP] Scheck, H.-J., P. Pistor, "Data Structures for an Integrated Database Management and Information

Retrieval System", : Proc. Inter. Conf. on Very Large Data Bases, 1982.

[S] Scholl M., : "Architecture pour le Filtrage dans les Bases de Donnees Relationnelles". These d'Etat, INPG, Grenoble, 1985.

[S+] Schweppe H. et al. : "RDBM - A dedicated Multiprocessor System for Database Management", Proc. of the 7th International Workshop on Database Machines, San Diego, September 1982, also in "Advanced Database Machine Architecture", D.K. Hsiao Editor, Prentice-Hall 1983, pp.36-86.

[Sh] Shemer J.L, Necks P.M. : " The Genesis of a Database Computer ", in Compute, Nov 1984, pp 42-56.

[Sl] Slotnick D.L., : "Logic per track devices", in "Advances in Computer", Vol. 10, Academic Press, New York, 1970, pp. 291-296.

[St] Stonebraker M.R, Wong E., Kreps P., "The Design and Implementation of INGRS " , ACM-TODS vol 1, no 3, Sept 1976.

[UNIX] Ritchie,D.,M., Thompson,K. :"The Unix Time Sharing System." Proc. ACM 4t SOSP CACM 17, no7, 1974, pp. 365-375.

[V] Verroust A., : "Characterization of Well-Behaved Database Schematas and their Update Semantics", Proc. Int. Conf. on Very Large Data Bases, Florence, 1983.

[VERSO] "Verso User Manual", INRIA Internal Report (to appear).

[Z] Zloof, M., "Query-By-Example: A Data Base Language" IBM Systems Journal 16, 1977, pp. 324-343.