The content-addressable page manager of Sabre.A multi -micro Data base computer

Georges Gardarin, P. Faudemay, Patrick Valduriez, Y. Viemont

INRIA

CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (3) 954 90 20

Rapports de Recherche

N° 284

THE CONTENT-ADDRESSABLE PAGE MANAGER OF SABRE

A MULTI-MICRO DATA BASE COMPUTER

Georges GARDARIN
Pascal FAUDEMAY
Patrick VALDURIEZ
Yann VIEMONT

Avril 1984

LE GESTIONNAIRE DE PAGES ADDRESSABLES PAR CONTENU DE SABRE, UN CALCULATEUR BASES DE DONNEES MULTI-MICROPROCESSEUR

Georges GARDARIN, Pascal FAUDEMAY

Patrick VALDURIEZ, Yann VIEMONT

Projet SABRE, INRIA, BP.105
78153 Le Chesnay-Cédex (France)

Université de PARIS VI, Institut de Programmation
Tour 55-65, 4 place Jussieu, 75230 Paris-Cédex 05 (France)

RESUME

Cet article décrit la méthode d'accès associative du calculateur bases de données SABRE. Cette machine emploie une méthode d'accès rapide et paramétrée ainsi que des processeurs spécialisés de filtrage. La méthode d'accès est basée sur une nouvelle structure, appelée arbre de prédicats, qui permet de spécifier d'une façon unifiée différentes fonctions de hachage multi-attributs. Une fonction de hachage est définie par un ensemble ordonné de prédicats de recherche à accélerer. Le dictionnaire qui décrit le placement par arbres de prédicats est organisé comme une relation qui peut être traitée efficacement par les filtres. Les filtres sont des processeurs spécialisés associés aux disques et réalisant les opérations de sélection sur les données pendant le temps de transfert disque. Les principaux avantages de la méthode d'accès associative sont sa généralité et son efficacité.

THE CONTENT-ADDRESSABLE PAGE MANAGER OF SABRE, A MULTI-MICRO DATA BASE COMPUTER

Georges GARDARIN, Pascal FAUDEMAY,

Patrick VALDURIEZ, Yann VIEMONT

Projet SABRE, INRIA, BP.105
78153 Le Chesnay-Cédex (France)

Université de PARIS VI, Institut de Programmation
Tour 55-65, 4 place Jussieu, 75230 Paris-Cédex 05 (France)

ABSTRACT

This paper reports on the content addressable access method of the SABRE database computer. This machine uses a fast and parametrized access method and filter processors. The access method is based on a new structure, called clustered predicate tree, which allows the user to specify in a unified way various multi-attribute hashing schemes. The hashing function is defined by a hierarchically ordered set of search predicates to be accelerated. The directory used for the implementation of clustered predicate trees is organized as a relation which can be searched efficiently by filters. Filters are special purpose processors associated with disks performing selection operations on data during transfer time. The main advantages of the content addressable access method are its generality and its efficiency.

.../...

1. INTRODUCTION AND BACKGROUNDS

A new class of data base management systems characterized by high level manipulation languages and based on the relational model is actually being commercialized [ASTR76, STON76]. More functionality is provided to the user. However, performance enhancement remains a severe problem. Several techniques are available to improve response time [STON83], among them pre-compiling repetitive queries [CHAMB81], improving certain operating system functions, using specialized purpose micro-computers for filtering data on the fly [HSIA83], storing tuples in a content addressable page manager [ROOM82], performing functions in parallel [DEWI83, VALD84], and so on.

In this paper, we report on the internal structure of the SABRE data base computer. The SABRE system [GARD83] herein is a software-oriented system being implemented on a french 68 000 based multi-micro processors, called the SM90 [FING81]. The first aim of the system is to get short response-time ; in particular, we would like to get all records satisfying most frequent queries in no more than two disk accesses when the data base is already active.

A main component of SABRE for reaching the two disk access principle [NIEV81] is the content addressable page manager, called MAPS (an achronism for a Memory which is Associative and Parallel for SABRE). MAPS uses two basic techniques to enhance performances :
- filter processors performing selection on the fly and working in parallel on several disk units,
- a parametrized fast access method for dynamic files which can be seen as an extension of extendible hashing [FAGI79] using filters and various hashing functions.

Filters are more and more attractive devices for performing selections in a data base computer. A filter is a specialized processor associated to a disk track, or a disk or a group of disks [DEWI81] receiving all data coming from the associated part of the secondary memory and moving only data satisfying a search criterion in the computer main memory. The main advantages of filters are :
- they perform selection on the fly during the transfer time from disks to memory, saving the selection processing time necessary in classical approaches,
- they restrict the amount of data to be moved in main memory, improving by this feature the buffering strategy,
- they improve the degree of parallelism for selecting data when associated to each disk unit.

Many filters have been designed [BANC80, HASK83, ROBE78, ROHM81, SU75...]. In the near future, we can envision very inexpensive VLSI based filters. MAPS includes specialized filters codding mono-attribute predicates with a finite state automaton to perform selection.

So far, our filtering algorithm is implemented by software on a specific processor, but a hardware version with microcode is being built.

Indexing and hashing are two fundamentally different techniques for implementing access methods to relations. In the former, a dynamic data structure is maintained and searched to compute a logical address. In the latter, an algorithm is executed to compute a logical address. Both techniques require a directory to perform the logical-physical mapping. In contrast, indexing generally implemented with balanced trees [BAYE72] adapt gracefully to the file content under insertion and deletion, while up to the late seventies, hashing was not addressing the issue of gradual adaptation of structure to fit the data. Balanced trees support random accessing based on a key value as well as sequential accessing of records by key order. Hashing only provides random accessing but generally with better performances than trees, since the data structure representing the hashing function is smaller than an index. Intermediate structures are now coming [LOME83] for merging the advantages of both techniques.

Over the past decade, new hashing schemes for structuring large dynamic files have been proposed. The first scheme of dynamic hashing was proposed for main memory tables in [KNOT71]. Then, three similar but distinct dynamic hashing techniques for files were proposed in [FAGI79], [LARS78] and [LITW78]. The most thorough approach is called extendible hashing [FAGI79]. With this method, a general hashing function f is associated to a file. Let K be the key of the file. Then the directory is addressed with the first d bits of the hashing function result f(K), where d is the file level. When a first bucket at level d is full, the directory size is multiplied by two, the saturated bucket is divided in two and the file level is increased to d+1. Then (d+1) bits of the hashing function are used to address the directory. The spliting process is repetitive, starting with a file at level 1.

Recently, two new schemes have been proposed which can be seen as extension of dynamic hashing to multi-attribute hashing : the grid file [NIEV81] is a symetric multi-key file structure where directory extension is done by partitionning search spaces in two, using alternate attributes. Multi-attribute linear hashing [OUKS83] is also an extendible hashing where, after using a first hashing function of d bits, the next expansions are performed along different attributes. Another variant of dynamic hashing called trie hashing was proposed [LITW81] ; under our definition, it is really an intermediate between hashing and indexing : an index organized as a trie is maintained. However, it is possible to merge the index with the directory and to use the trie as a logical address computation support and therefore trie hashing can be seen as a true hashing.

In the SABRE context where associative search is performed by filters, dynamic hashing appears as an attractive technique for reducing search spaces. The main advantages are the following :

- it does not require index updating when tuples are inserted or deleted,
- some variants support multi-attribute access paths,
- a record satisfying a fully specified query (with all the hashed attributes specified) can be retrieved in at most two disk accesses : the first one to the correct part of the directory, the second one to the data bucket [NIVE81].

.../...

A problem with hashing is that many variants have been proposed, each of them with different hashing functions and/or directory organizations. Therefore, we looked for a generalized and parametrized method allowing the system to implement different kinds of hashing as extendible hashing, trie hashing and multi-attribute hashings. We come out with a new data structure called "clustered predicate tree". This structure can be seen as a generalization of extendible hashing with :
. an associative directory which can be searched by filters,
. a hashing function parametrized by a predicate tree, each level of the tree being defined by a list of disjoint predicates.

In this paper, we present the internal design of MAPS which is therefore based on clustered predicate trees and filtering on the fly. In the next section, the objectives, the functionalities and the architecture of MAPS are summarized. Section 3 is devoted to the introduction of the new access technique based on predicate trees : it is shown how this technique generalizes all mono and multi-attributes dynamic hashing. Section 4 and 5 report on the search algorithm for partially specified queries : section 4 shows the use of the directory and section 5 describes the filtering technique.

2. OBJECTIVES, FUNCTIONS AND ARCHITECTURE OF MAPS

2.1. Objectives and functions :

MAPS achieves four basic objectives :
- to supply a content addressable paged and relational memory,
- to execute as much as possible each individual selection in parallel, as fast as possible,
- to solve concurrency conflicts between transactions,
- to guarantee transaction atomicity.

In this paper, we only report on methods and algorithms used for achieving the two first objectives.

The content addressable memory is first invoked mainly by a retrieval request. A retrieval request contains several arguments :
(1) a source relation name which specifies the segment in which the relation to retrieve is stored,
(2) a restriction expression that identifies, by content, which tuples are to be retrieved,
(3) a projection specification that determines attributes to be returned from the selected tuples,
(4) a target relation name which determines where to store the result as a relation.

In addition to the resulting tuples in the target relation, a retrieval request returns three parameters : the sizes of the target relation in number of pages and in number of tuples, and a response code specifying possible errors. The general form of the retrieval request call is as follows :

Response := SELECT (transaction-id, source-relation-id, target-relation-id, restriction-expression, projection-specification, page-number, tuple-number)

where the return parameters are underlined, and transaction-id is the identifier of the transaction for which the request is run.

The target relation is a temporary relation, that is one which will be destroyed at transaction end ; we can apply SELECT on a temporary relation. For any relation including temporary relations, the client can read the relation content page per page, either in sequential mode or in direct mode. Reading a page is performed by the following primitive :

Response := READ (transaction-id, relation-id, page-number, buffer)

Two update facilities are offered. To update a relation, the user must at first either retrieve tuples in a temporary relation, or insert them in a temporary relation using the following primitive :

Response := INSERT (transaction-id, relation-id, tuples),

The temporary tuples can then either be append to another relation (generally a permanent data base relation), or deleted from another relation. This is done by using the following calls :

Response := UNION (transaction-id, source-relation, target-relation)

Response := DIFFERENCE (transaction-id, source-relation, target-relation)

The five primitives SELECT, READ, INSERT, UNION and DIFFERENCE are the only one offered by MAPS to the client to manipulate relations. However, four other requests allow the system to manage transaction. They are :

Response = BEGIN-TRANSACTION (transaction-id, user-id)
Response = READY-TO-COMMIT (transaction-id)
Response = COMMIT (transaction-id)
Response = ABORT (transaction-id)

which are the classical primitives required to start, commit and abort transaction in a distributed system environment [GRAY81].

Let us point-out that no primitive is necessary to create or destroy a relation : a relation is created when its identifier appears the first time ; it is destroyed either at transaction-end if it is a temporary relation, or when it becomes empty if it is a permanent relation.

2.2. The architecture of MAPS :

The hardware configuration of the machine which supports MAPS is portrayed figure 1. It is built with off-the-shelf hardware distributed by a french company. The processors are 68 000 and the machine is called SM 90 [FING81]. To each disk unit is associated a disk controller and a filter processor (FIP). This processor runs the algorithms we will describe in section 5. All filter processors can work in parallel and bring filtered data (that is, data resulting from a SELECT command and belonging to the target relation) into the associated memory page frames. The whole set of page frames is addressable by all processors and managed by a specialized processor : the cache memory processor (CMP). This processor receives all the commands from the clients of MAPS and sends the resulting tuples in response to a READ command ; it is also responsible for dispatching subcommands to the filters. The communication bus is used to transfer pages and commands. Its speed is five megabytes per second.

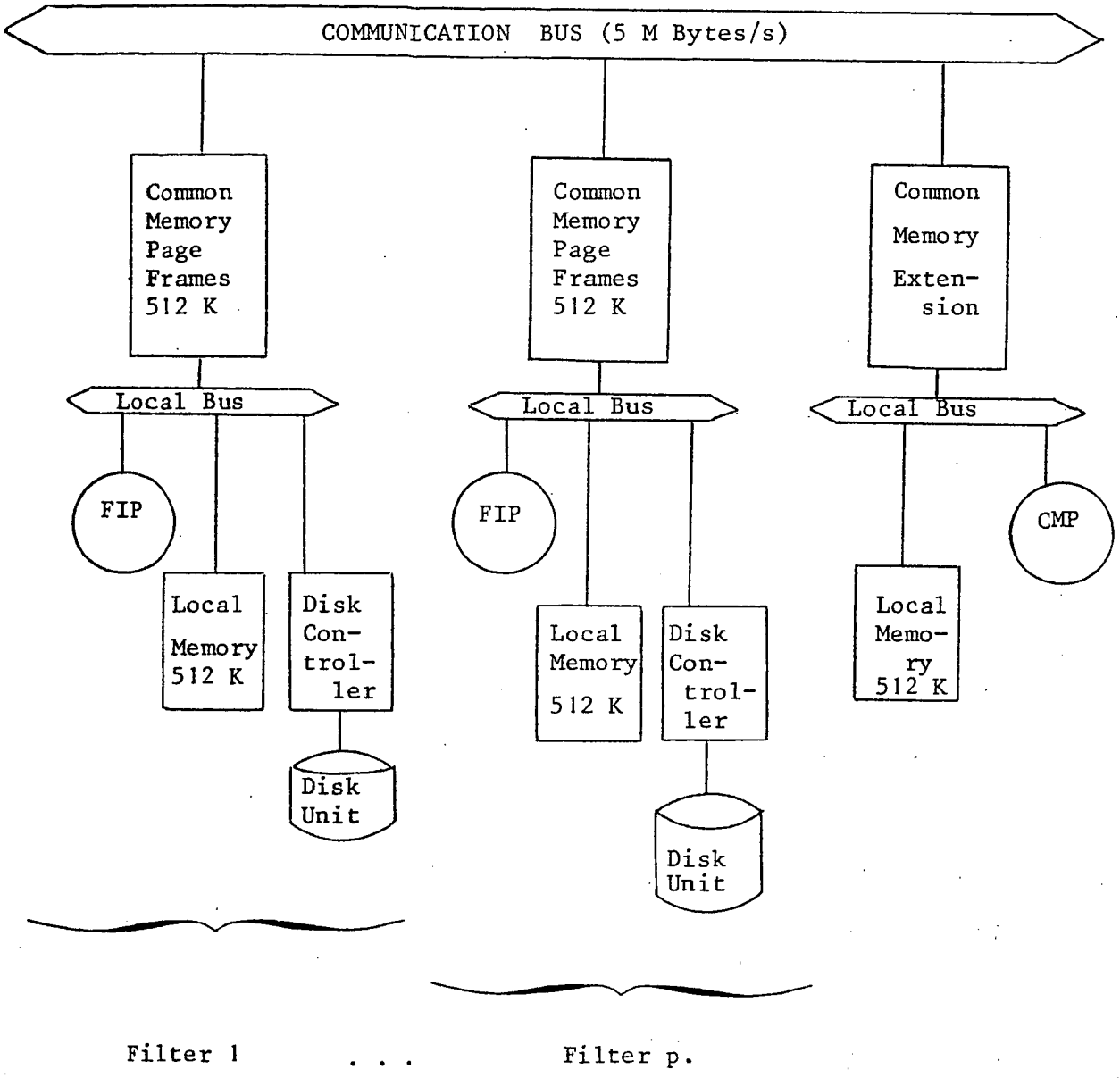


Figure 1. : The hardware architecture of MAPS

3. THE PLACEMENT OF TUPLES ON DISKS

3.1. Relation fragmentation with predicate trees :

Tuples are stored inside associative partitions [BANE78]. The size of such a partition is generally a disk track but can be less : it is a system generation parameter. The partition is the unit of filtering. A filter processor can retrieve all tuples inside a partition satisfying a boolean expression, and can bring selected attributes of the tuples into main memory. Also, a filter performs the insertion of new tuples inside a partition and the deletion of previously selected tuples if requested. The addressing inside a partition is done by content, using boolean expressions. The algorithms to perform filtering are presented in section 5.

Relations are dynamically divided into fragments. A fragment is a set of 1 to q partitions (where q is generally a multiple of the number of parallel filters) in which tuples are searched by filtering in parallel the partitions. It is also a part of a relation which corresponds to a bucket in classical hashing methods. When a fragment reaches q partitions and new tuples have to be inserted into the fragment, it is split into two fragments according to a splitting policy. Thus, MAPS limits the space in which tuples are searched by filters to f partitions.

For each relation R, the description of the criteria to group tuples into fragments is done by a predicate tree (PT). A predicate tree of depth p, is a balanced tree where each internal node of level (i-1) has mi sons. Each level i of the tree is associated to mi predicates Pi1, Pi2 ... Pimi. At level i, the jth sons of nodes of level (i-1) correspond to the same predicate Pij. The number mi of predicates defining level i is called the branching factor of level (i-1).

A predicate Pij is a boolean combination of atomic predicates expressed in disjunctive normal form. Each atomic predicate is of the form :

$$\langle \text{FUNCTION (ATTRIBUTE)} \rangle . \langle \text{OPERATOR} \rangle . \langle \text{VALUE} \rangle$$

where the function is either the identity function (no function specified) or a hashing function and the operator chosen among $\{=, >, \geqslant, <, \leqslant, \neq\}$. A predicate Pij can be thought as a restriction predicate over relation R. The predicates defining a level must satisfy the following assertions :

(1) they must partition R into mi disjoint sub-relations Rij, i.e. for all j, for all k such that $j \neq k$ $\quad Rij \cap Rik = \emptyset$

(2) any tuple of R must belong to one (and only one according to (1)) of the mi sub-relations Rij, i.e. $R = \bigcup_j Rij$.

In addition, we say that a predicate tree is consistent if each path from the root to a leaf corresponds to a set of predicates which can all be satisfied by at least one tuple.

For example, let us consider a WINE relation having attributes vintage, year, area, degree and color :

WINE (VINTAGE, YEAR, AREA, DEGREE, COLOR).

A predicate tree of depth 2 for the WINE relation is represented figure 2.

.../...

Level 1 corresponds to predicates :
{(DEGREE <12) ; (DEGREE ≥ 12).
Level 2 corresponds to predicates :
(AREA = "BORDEAUX") ; (AREA = "BOURGOGNE") ;
(AREA $\neq$ "BORDEAUX" AND AREA $\neq$ "BOURGOGNE") .

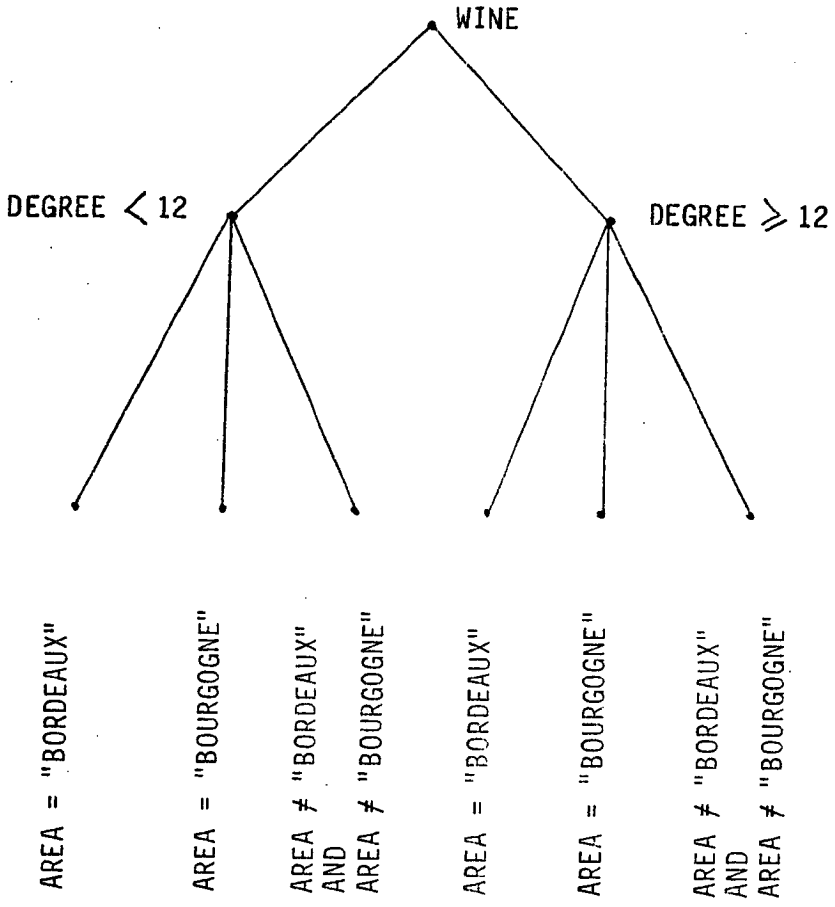


Figure 2. : An example of a PT

A predicate tree as defined above can be utilized to cluster tuples by assigning a fragment to each leaf. Each tuple in a fragment satisfies the intersection of all the predicates on the path from the root to the leaf corresponding to that fragment. As we defined the predicate tree as a balanced tree where each level is developed, assigning a fragment composed of at least one partition would lead to an under occupancy of the memory,

since some leaves can be empty in general. Therefore, underoccupied leaves of a same father are grouped together in the same fragment. If all the sons of a father are grouped in a single fragment, then the father becomes itself a leaf. This process of grouping underoccupied fragment can occur until obtaining an overloaded fragment, according to the maximum number of partitions q. This mechanism enables to cluster tuples verifying the same predicates and to maintain a reasonable lower bound on average partition occupancy.

The tree obtained by associating fragments with certain nodes of a predicate tree is called a clustered predicate tree. We can define more precisely a clustered predicate tree. For each node, let us call path predicate the predicate which is the intersection of all predicates from the node to the root. Considering any node in a predicate tree, we can define the weight of that node to be the size of the tuples (expressed in number of partitions) which satisfy the path predicate. Then a clustered predicate tree of order q is a tree deduced from a predicate tree by leaving out all nodes whose father has a weight less than q. This process for constructing a clustered predicate tree of order 3 is illustrated figure 3 on the wine relation. Such clustered predicate trees are then used to fragment relations, a fragment being assigned to each leaf.

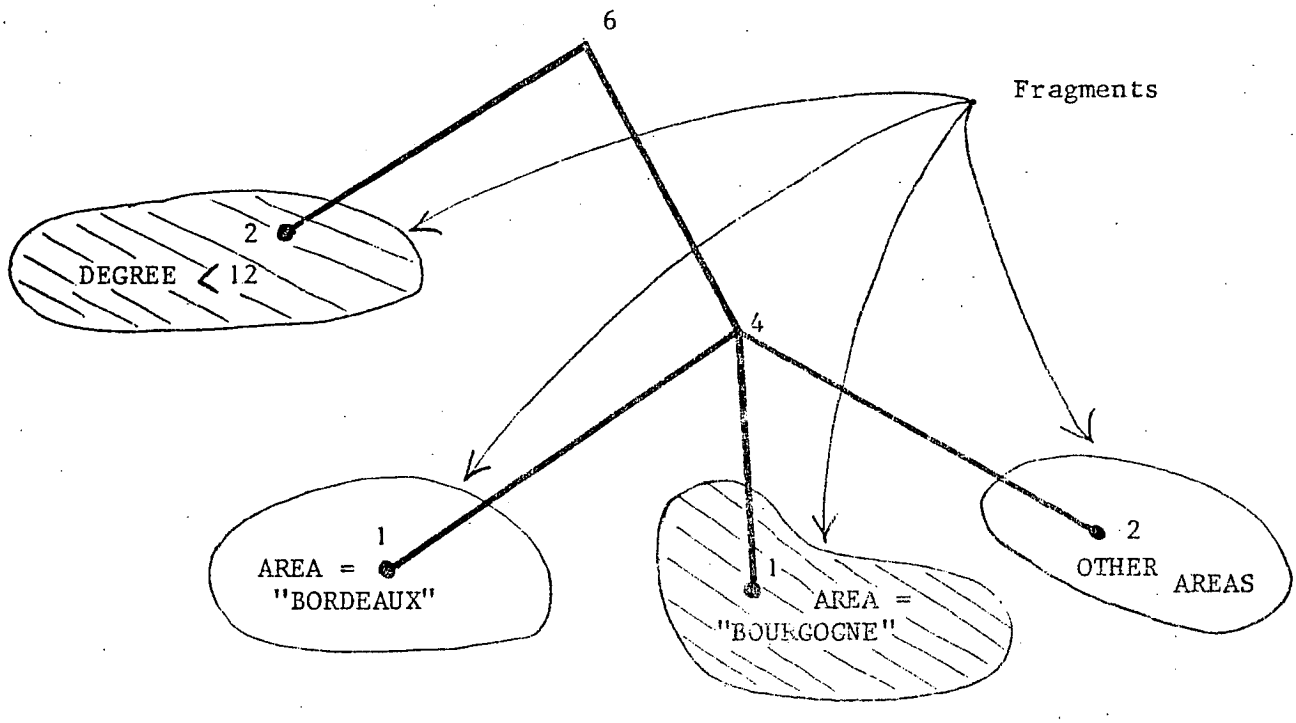


Figure 3. : Constructing a clustered predicate tree of order 3

3.2. Predicates trees as a generalization of hashing schemes :

A predicate tree can be seen as a data structure used by a hashing algorithm to compute the logical address of the fragment (i.e. the bucket) in which a tuple is placed. We shall see in the next section how to determine the physical addresses of the fragment partitions through a directory. Because the PT is predefined at relation creation, the implemented method is a hashing method in which the address computation is parametrized with a tree structure well suited for an non-procedural query language : the predicate tree.

When changing the PT which parametrizes the hashing method, it is possible to generate many known mono or multi-dimensional hashing schemes. For example, we can implement variants of the following schemes :

. All classical mono-attribute hashings without overflow using p buckets are obtained by defining a one level PT containing the following predicates :
{h (K) = 0 ; h (K) = 1 ; ... h (K) = p-1} where h is the hashing function and K the key. This tree is portrayed figure 4.

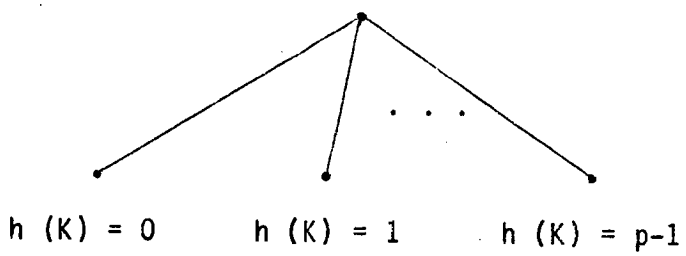


Figure 4. : PT for classical hashing

. Classical mono-attribute hashings with overflow managed by rehashing over d buckets are obtained by a two level tree as portrayed figure 5. With such hashings, only tuples inserted in a saturated fragment are generally distributed at the second level.

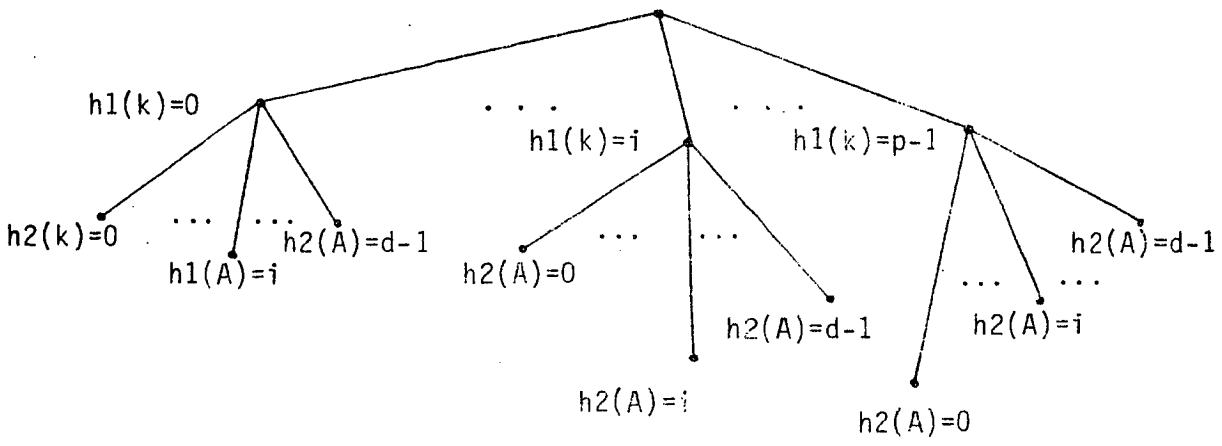


Figure 5. : PT for hashing with rehashing overflow

.../...

. Extendible hashing [FAGI79] can be defined by a family of hashing funtion h0, h1, h2 ... hi derived from an initial and universal hashing function h satisfying the following conditions :

$h : A \rightarrow \{0, 1, \ldots 2^N-1\}$

$ho : A \rightarrow \{0, 1, \ldots 2^P-1\}$
where $N >> P$ and $ho(A) = h(A)/2^{N-P}$

$hi : A \rightarrow \{0, 1\}$
where $hi(A) = BIT_{p+i}\ (h(A))$

BIT_j (X) means the j th bit of the bit string x starting from the left. Function ho considers the first P bits of h(A) while hi considers one more bit than hi-1. P is the number of branches at level 0 (in other words, the number of initial buckets). When a fragment of level i is full, it is split in two using the next function hi+1. Therefore, the extendible hashing can be obtained by the predicate tree of figure 6.

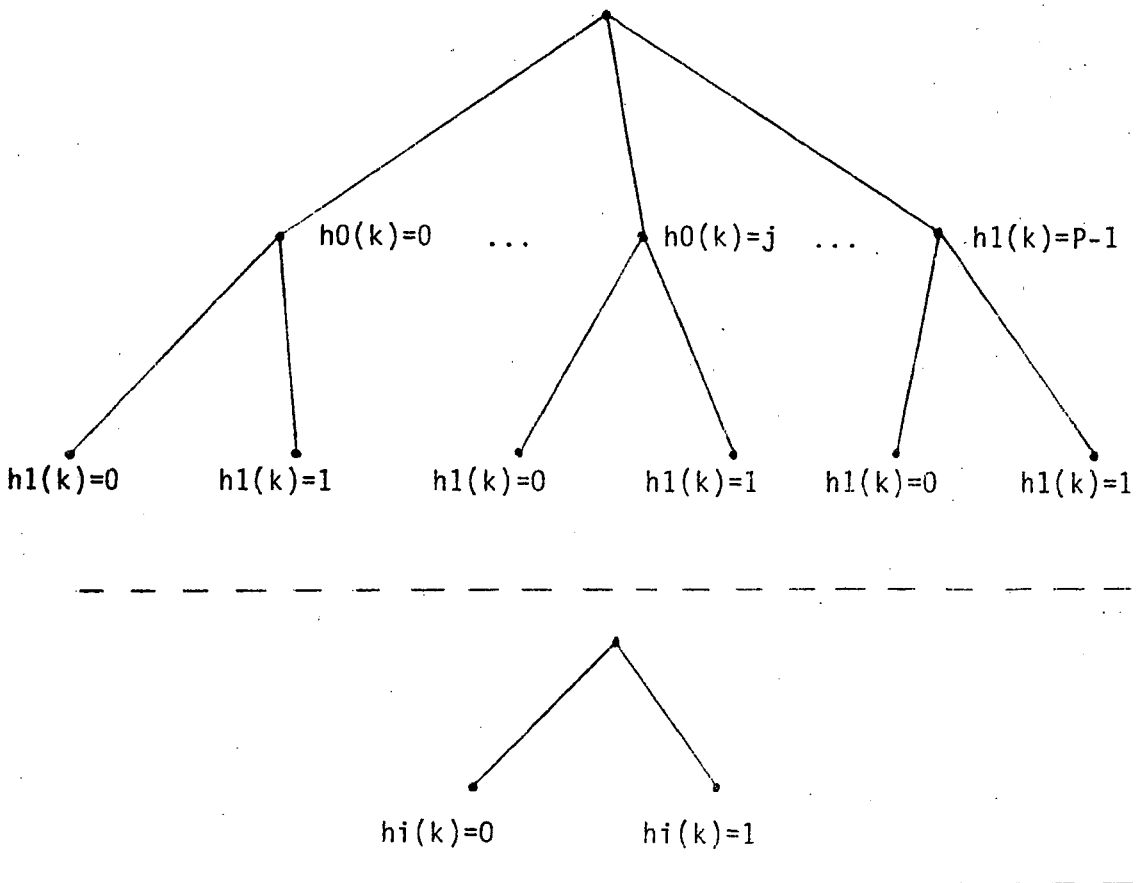


Figure 6. : PT for extendible hashing

. Virtual hashing [LITW78] and linear hashing [LITW80] use exactly the same PT ; only the splitting policy is different. We use a splitting policy similar to extendible hashing, consisting in splitting in two a saturated fragment, as we shall see latter : it is the simplest and the most robust policy.

. Multi-attribute linear hashing [OUKS83] is an extension of linear hashing where the attributes used for defining predicates at each level are changing as follows :

- h0 is a function of attribute A0
- h1 is a function of attribute A1
- h2 is a function of attribute A2
  ...
- hi is a function of attribute Aj where j = (i-1) modulo d, d being the number of attributes used for hashing.

In addition, the splitting policy used is that of linear hashing, that is a pointer indicates the fragment to split when one overflows.

. Trie-hashing [LITW81] or at least one of its variations can be obtained by associating each level of the predicate tree to a character of the key, as portrayed figure 7. A variation of trie-hashing is a binary radix tree hashing where each level is associated to one bit of the key.

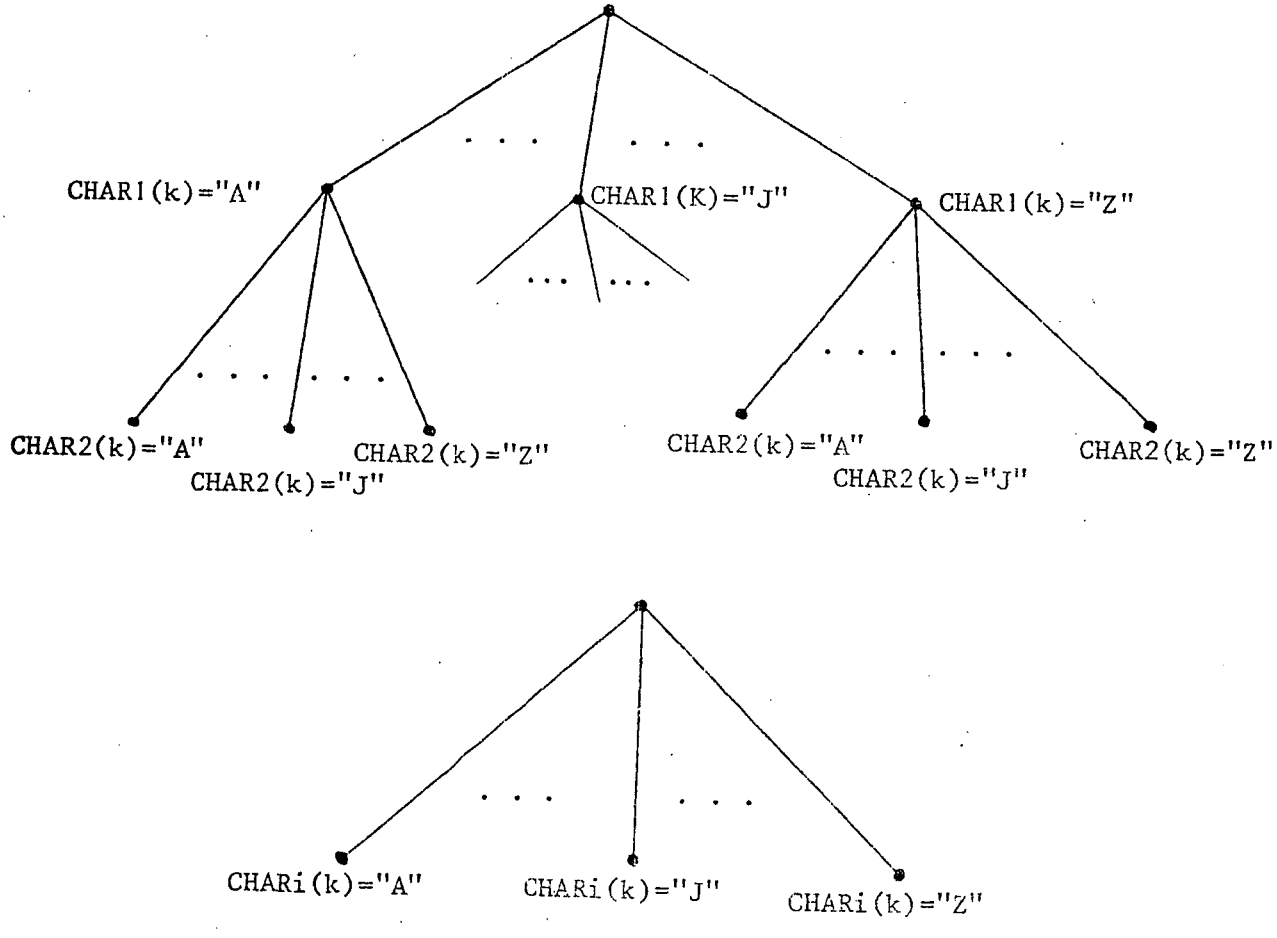


Figure 7. : PT for trie-hashing

. The grid-file [NIVE81] is a multi-attribute scheme where fragments are split in two according to a cycle of attributes. Let us define it for the two dimensional case where the cycle of attributes is { A0, A1 }. The generalization to n attributes is straightforward. The predicate tree is a binary predicate tree where each node has two sons. The attributes { A0, A1 } appears cyclically at each level. Thus, let { f2i (A0) = 0 ; f2i (A0) = 1 } be the predicates associated to level 2i for i = 0, 1, 2 ... and let { f2i+1 (A1) = 0 ; f2i+1 (A1) = 1 } be the predicates associated to level 2i+1 for i = 0, 1, 2 ... We are going to define f2i and f2i+1 below. The predicate tree associated to the grid-file is portrayed figure 8.

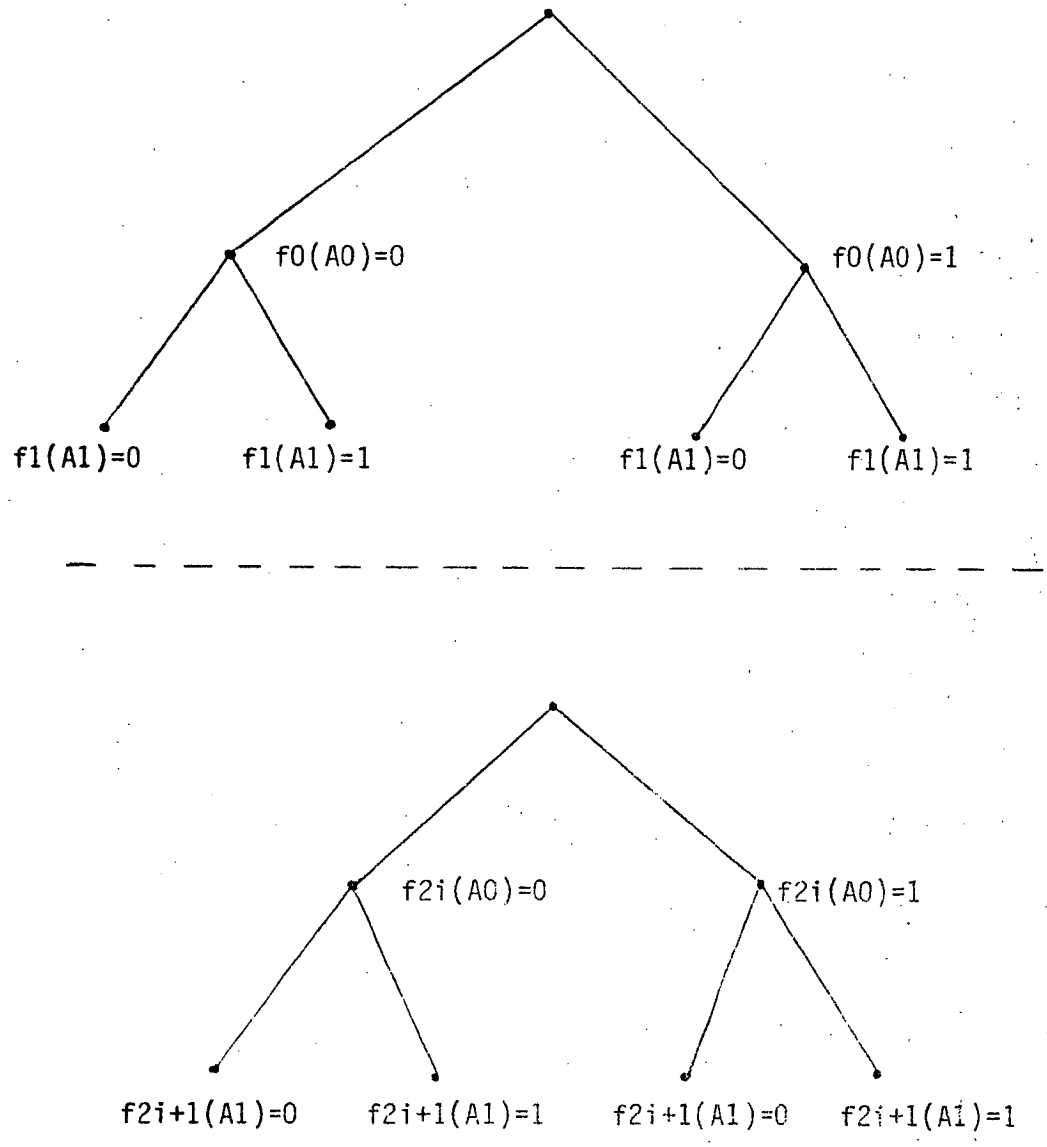


Figure 8. : PT for the grid-file

.../...

The only problem is now to determine f2i and f2i+1. According to the grid file specifications, at each split, the interval of the considered attribute (A0 or A1) is divided in two intervals separated by the median value. Thus, f2i and f2i+1, are given by the following formulas :

F2i(Ao) = E [2**(i+1)*(A0-MIN(A0))/(MAX(A0)-MIN(A0))] modulo 2

F2i+1(A1) = E [2**(i+1)*(A1-MIN(A1))/(MAX(A1)-MIN(A1))] modulo 2

where MIN and MAX are respectively the minimum and maximum values of the argument attribute, E(a) means the entire part of a, and modulo 2 is the rest in dividing by 2 the integer E(a). To explain the formulas, it is sufficient to say that the f part between brackets is a real which determines the rank of the associated branch in the binary predicate tree at the considered level, going from the left (branch 0) to the right (branch 2**(i+1)).

Probably a large number of possible predicate definitions is useful for certain applications, with various degrees of generality and complexity. The implementation of PTs in SABRE is actually restricted to a minimum set of predicate definitions, chosen for the following criteria : the specification of a level must be compact and simple because a PT must fit easily in memory and have a low processing cost ; the minimum set of predicates should permit to represent combinations of various known methods. The main restriction is that a level consists of predicates defined on the same attribute. This facilitates the definition of a level by a database administrator and the management algorithms. Multikey predicates are specified with several levels. The following level definitions are provided :

(1) Classical hashing : hashing functions such as f(K) = k mod P are allowed.

(2) Digital hashing : the function CHARi(K) gives the rank of the ith character (if k is alphabetical or alpha-numerical) or the ith digit (if numerical key) in its alphabet (A..Z , 0..9 , etc). As an example, it permits to represent trie-hashing where the same key is involved at each level using a different character of the key.

(3) Interpolation hashing : this function is similar to that of [BURK83] and not far from that of the grid file. Giving the minimum and maximum values of a key and the branching factor m, the function is specified by :

f(K) = E [m*(K-min(K))/(max(K)-min(K))]

It generates intervals by uniform partitionning of the space.

(4) Enumeration of a domain : the list of possible values of a key domain is enumerated. The key word OTHERS can be given by the designer of the PT to inform the system that a value not in the list can appear. If this word is not employed, the domain is completely defined ; that allows the PT to implicitely store integrity constraints on this domain. This feature will be seen to be useful in the next section.

(5) Enumeration of intervals : if interpolation is not wished because of a totally non uniform distribution of values of data, it is then possible to enumerate the limits of the intervals in key order. Two key-words (SMALLEST and GREATEST) can be optionnally used, meaning the possibility of a value smaller than the minimum given or greater than the

maximum given. As an example, the following list {smallest, 10, 30, 50} defines the predicates

$K < 10$, $10 \leqslant K < 30$, $30 \leqslant K < 50$

This also implements in a simple way integrity constraints on a domain.

3.3. The directory organization :

In this section, we describe in more details the directory which allows the system to map a logical fragment address to physical addresses. Its most important characteristic is associativity.

A logical address called a signature of a tuple is the address of the leaf of the PT corresponding to that tuple. To address a node in a PT, we label each branch issued from an internal node by the rank of the corresponding predicate in the list of predicates associated to the level. The address of a node is then the bit chain generated by the list of labels going from the root to the node. If we consider a clustered predicate tree, it is possible to define the signature of a fragment as the logical address of the corresponding node in the (clustered) PT. Figure 9 gives some signatures of tuples and fragments using the PT defined in figure 2 and the clustered PT of figure 3. Predicates are labelled by levels as indicated in the predicate table.

PREDICATES

LEVEL	LABEL	PREDICATE
1	1	DEGREE $<$ 12
1	2	DEGREE $\geqslant$ 12
2	1	AREA = "BORDEAUX"
2	2	AREA = "BOURGOGNE"
2	3	AREA $\neq$ "BORDEAUX" AND AREA $\neq$ "BOURGOGNE"

SIGNATURES

TUPLE	TUPLE SIGNATURE	FRAGMENT SIGNATURE
VOLNAY-1978-BOURGOGNE-13-ROUGE	2-2	2-2
JULIENAS-1980-BEAUJOLAIS-13-ROUGE	2-3	2-3
MEDOC-1981-BORDEAUX-11-BLANC	1-1	1

Figure 9. : Examples of signature

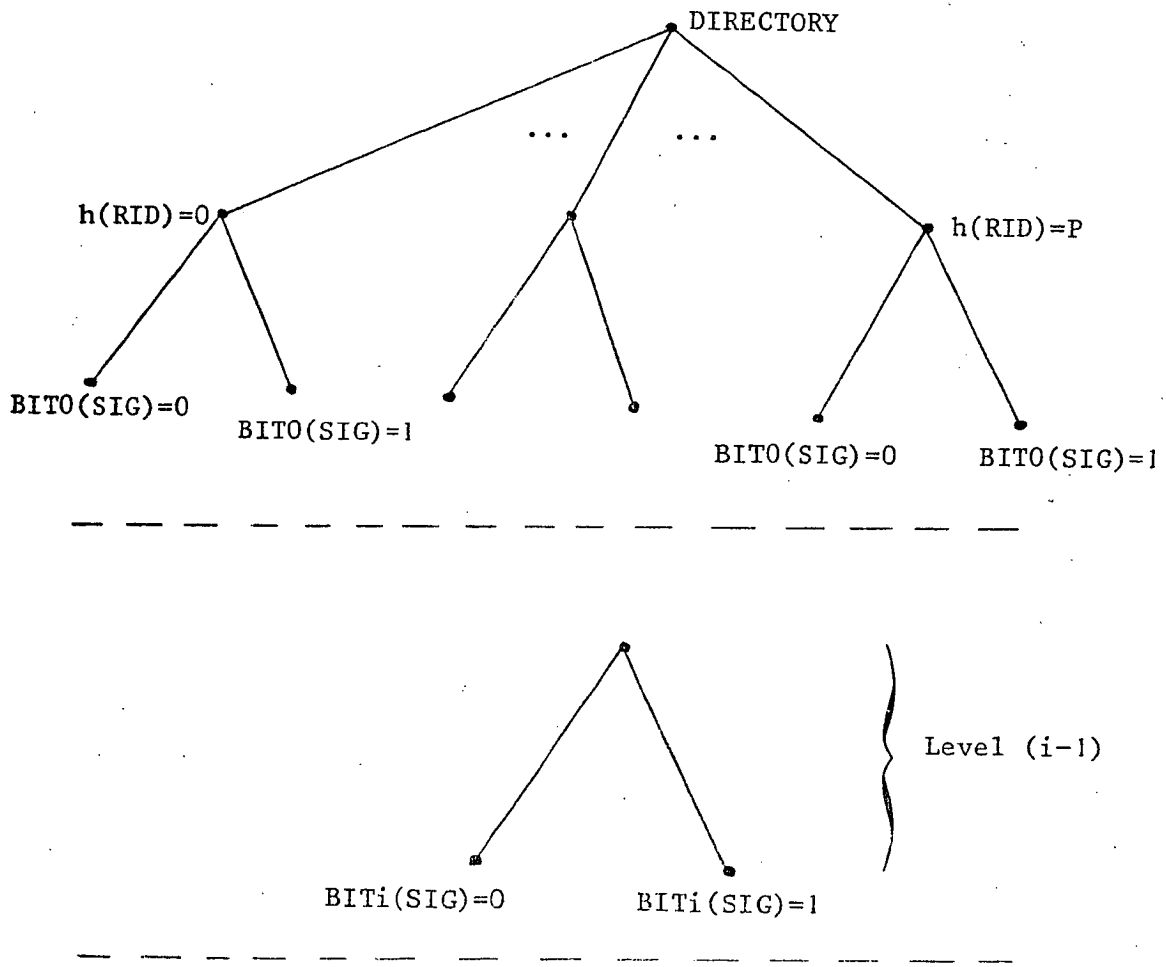
The directory associates to each fragment signature the physical addresses of partitions containing that fragment. All directory entries are stored in a unique relation DIRECTORY whose schema is :

DIRECTORY (RELATION-ID, FRAGMENT-SIGNATURE, PARTITION ADDRESSES)

Using a unique relation for the directory gives two major advantages :

- filters can be employed to search for a signature or a part of a signature as shown in the sequel,
- the directory can itself be partitionned in fragments using a predefined PT which is portrayed figure 10. As this PT and its directory stay in main memory, it is possible to guarantee that the part of the directory to scan for finding a given signature is no more than one fragment of one partition.

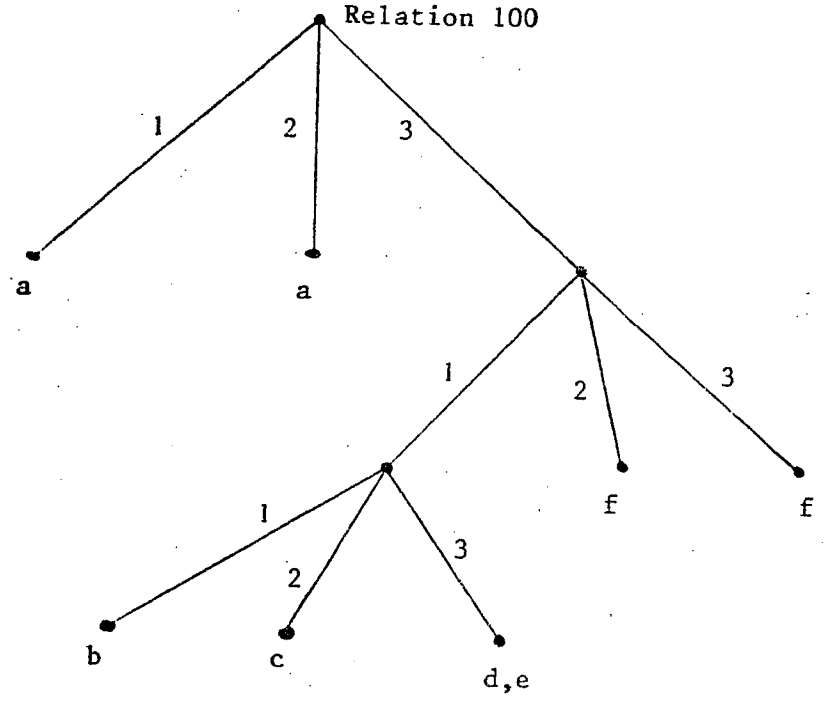
Thus, we enforce the two access principle of [NIVE81].



- h is a hashing function (the modulo P function)
- BITi gives the bit i of the argument
- SIG = signature
- RID = relation-id.

Figure 10. : The PT of the directory

In summary, figure 11 portrays a clustered predicate tree and the part of the directory corresponding to that tree. The PT associated to that relation (number 100) is a balanced tree with three predicates per level, having 27 leaves.



DIRECTORY	Relation-id	Signature	Addresses
	100	1	a
	100	2	a
	100	311	b
	100	312	c
	100	313	d,e
	100	32	f
	100	33	f

Figure 11. : A clustered predicate tree and the associated directory part

3.4. Insertion and deletion of tuples :

Let us first consider the insertion of tuples in a relation whose clustered predicate tree is of order q. The partition address in which a tuple must be inserted is determined by looking for the tuple signature in the directory : this problem will be addressed in the next section. Let us study here the placement problem.

Let us assume that the relation is empty. Then, the first tuples are inserted in a single partition allocated to the relation. A problem arises when the partition is full. If the order q of the clustered PT is greater than 1, then a new partition is just allocated to the fragment. This partition is allocated if possible on a parallel disk in order to maximize parallelism when searching. The real problem arises when q partitions have been allocated, that is when the root fragment is saturated.

More generally, when a fragment at level (i-1) is full, it is split. A splitting policy could have been to partition the fragment according to the mi predicates of level i of the predicate tree. Then, the clustered PT would grow by addition of mi branches issued from a saturated node. This solution would have lead to a very low load factor for newly allocated partitions (for example, with q = 3, mi = 30 we would obtain 30 partitions occupied at 10 %).

Finally, the solution which has been chosen consists in splitting the fragment in two, using the next bit of the signature. Therefore, all branches having the first bit (on the left) at 0 are grouped in one fragment and all others in another fragment. The signature in the directory grows one bit at a time. When all bits corresponding to a level have been developped, the initial fragment will have been split in mi fragments, but in a progressive manner and only if necessary. Thus, at each new fragment creation, the load factor of newly allocated partition is (q/2) / (E (q/2) + 1) (for example, with q = 3, we could obtain 4 partitions occupied at 75 %). The approach for splitting has certain similarities to that of DL-trees [LOME83].

What happens if a clustered PT of order q overfills, that is a leaf of the PT becomes loaded with more than q partitions ? In that case, all the bits of the tuple signature have been expanded : it is not possible to expand more the fragment signature. We use then a very simple overflow strategy which just consist in adding partitions to the pending fragments ; thus, in that case, we tolerate fragments with more than q partitions. In theory, it could be possible to add more levels to the predicate tree to split again overloaded pending fragments.

The inverse operation of insertion is deletion. While insertion leads to fragment splitting, deletion of tuples leads to fragment grouping. When two brother fragments appear at the same level (with signature of equal length differing only by the last digit), it is souhaitable to merge these two fragments as soon as their tuples can fit in q partitions. Whenever one of the two fragments in which we just delete a tuple appears to have a unique partition whose load factor is less than a given treshold (0.4), the system tries to merge the fragment with its brother. Thus the clustered predicate tree shinks as tuples disappear.

4. THE RETRIEVAL OF PARTITIONS CONTAINING QUALIFIED TUPLES

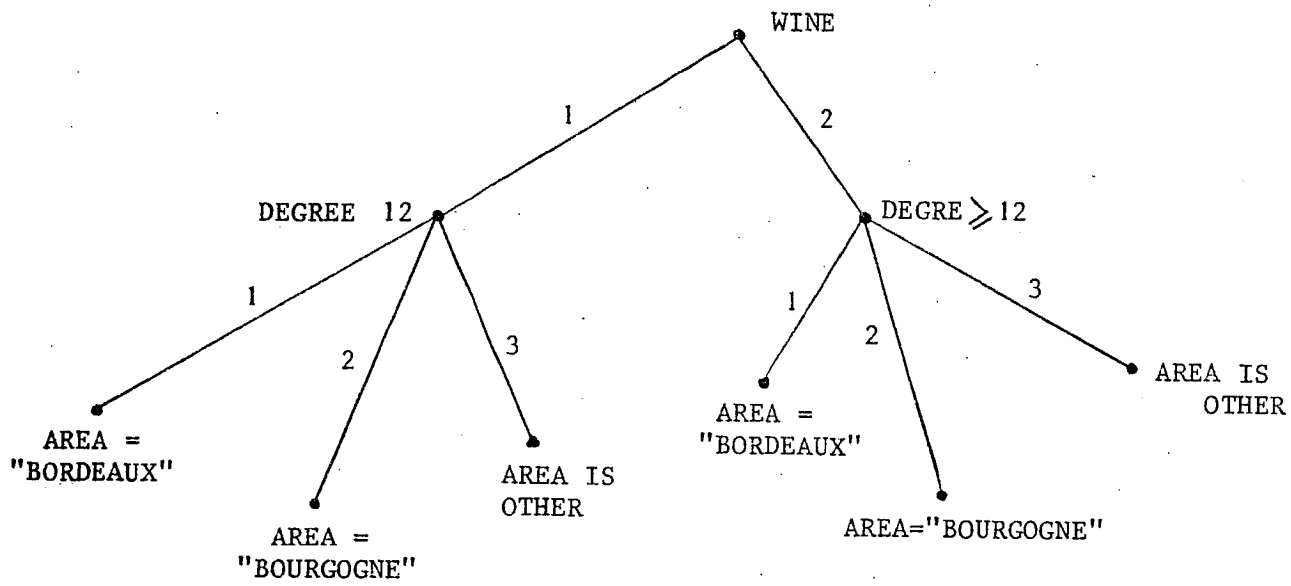
The retrieval of tuples verifying a selection qualification is performed in two steps :

(1) Physical addresses of partitions which may contain pertaining tuples are first retrieved by a filtering of the directory.
(2) Pertaining tuples are then retrieved by filtering the partitions whose addresses have been obtained as a result of step 1.

This section develops step 1.

4.1. Selection qualification encoding :

To perform the selection in the directory, we first encode the selection qualification as a list of signature profiles. A signature profile is an incomplete signature where missing levels of branch numbers have been replaced by the unknown value, here denoted ".". For example, figure 12 portrays some signature profile lists resulting from encoding some qualifications with the given predicate tree associated to the wine relation.



QUALIFICATION	PROFILES
DEGREE < 12	1 .
(DEGREE < 12) AND (AREA = "BOURGOGNE")	1 2
AREA = "BOURGOGNE"	. 2
DEGREE < 11	1 .
DEGREE < 13	1 . , 2 . (.. is an alternative)

Figure 12. : Examples of signature profiles

A general algorithm which encodes a selection qualification is now presented. Let Q be the qualification, Q is compared with the PT associated with the relation R in order to give one of the following result A :

- A = R, each tuple of R can satisfy Q
- A = $\emptyset$, no record of F can satisfy Q
- A = $\{Rij / Pij \wedge (Pij \Longrightarrow Q)\}$

In the latter case, the algorithm generates a set of signature profiles which determine the fragments Rij containing candidate tuples to match Q. The principle of the algorithms is to select in the PT the predicates not contradictory with Q. Two predicates P and Q are said contradictory if, for any interpretation of P, P $\Longrightarrow$ not Q. For example, predicates DEGREE >11 and DEGREE <10 are contradictory while DEGREE >11 and DEGREE >12 are not. When comparing the PT with Q, two cases can occur. The first one is that a level i is defined on attribute Ai which is not involved in Q. Thus, each predicate of level i can be satisfied and each node address is selected. The unknown value is generated in the signature profile part.

The other case is that level i is defined on an attribute involved in Q. Then, for each predicate Pij not contradictory with Q, j must be kept as a candidate signature profile part. If no such j is found and no signature part already exists for level i, then A = 0 and the relation will not be accessed. This occurs when Q violates an integrity constraint implemented on a domain by the PT. A set of profile parts (different node addresses for a same level) can be reduced by setting to the unknown value a bit which appears to be "1" in a part and "0" in another one. Finally, the cartesian product of all signature profile parts determines the set of signature profiles to answer the query.

The algorithms which transforms a qualification Q into a set of profiles L proceeds in four main steps. Q is expressed in disjunctive normal form :
Q = (Q11 AND Q12 ... AND Q1k) OR ... OR (Qn1 AND Qn2 ... AND Qnkn)
where Qij has the form (A θ value), A being an attribute of the file and θ a comparison operator. A similar algorithm could be found for a qualification expressed in cunjunctive normal form.

The algorithm is portrayed in figure 13, where sets Li are supposed initialized to empty, noted []. The first step projects Q on the attributes involved in the PT. A boolean value is affected to each predicate Qij and set to true if the attribute of Qij is not in the PT and set to false otherwise. Then, simplification rules "P $\wedge$ true is replaced by P" and "P $\vee$ true is replaced by true" are applied. This avoids to test in the next step for disjunctions set to true. In particular, if Q becomes set to true then the whole relation must be searched, the profile being with all ".". The second step selects the predicates in the PT which are not contradictory with a cunjunction having not been set to true. For a cunjunction Qk composed of predicates on various attributes, levels corresponding to attribute involved in Qk are selected. For a level i selected, if the set of profile parts Li is empty, it is set to [none]. Then, the hashing function of level i is applied to the value in predicates on Ai of Qk and each j such as Pij is not contradictory with Qk is kept as a possible signature part.

The third step initializes all sets of profile parts Li, of which attribute Ai is not involved in Q (after simplification), to the element [.] meaning all possible values. If a set Li is still set to "none" then L is set to "none" and the algorithm terminates. The last step applies the cartesian product (noted X) to all sets Li giving the final set of signature profiles answering the qualification.

```
procedure PROFILE (Q : qualification ; L : set of profiles) ;
   L1, L2, ... Li, ... Lp : set of profile part ;
begin
simplify (Q) ;
for each remaining cunjunction Qk do
   for each attribute of Qk in the PT do
      begin
      i := level of the attribute ;
      if Li = [] then Li := [none] ;
      for j := 0 to mi-1 do
         if Pij ∧ (Pij ⇒ Qk) then Li := Li ∨ [j] ;
      end ;
for i := 1 to p do
   begin
   if Li = [none] then L := [none] ; exit ;
   if Li = [] then Li := [.] ;
   end ;
        p
L := ⨯   Li ;
       i=1
end ;
```

Figure 13. : Profile generation algorithm

4.2. Filtering of the directory :

With the signature profile list corresponding to a qualification, it is necessary to access the directory in order to get pertaining partition addresses. The first action is done in memory to localize the directory of the directory. Then, the partition address(es) (in general, one is sufficient) of the directory corresponding to the relation and the profile list is extracted from the directory of the directory. Finally, the partition(s) of the directory is (are) scanned either in memory if already in, or on disk with the filter.

Signature profiles are generated using the PT and not the clustered PT. Therefore, it is necessary to select in the directory all signatures which match a prefix of a signature profile.
For example :
- 1 matches profiles 1 . , . 2
- 12 matches profiles 12, . 2, 123

Comparisons of signatures and signatures profiles is done by the filter, using a special operator "prefix" denoted <>. At last, the partitions addresses are obtained by projecting the tuples of the directory on the "partitions" attribute, during the selection by prefix. Figure 14 illustrates signature profiles and prefix selection in a possible directory for the WINE relation (relation-id = 100).

DIRECTORY	RELATION-ID	SIGNATURE	PARTITIONS
	100	1	a,b
	100	21	c
	100	22	d
	100	23	e,f

QUALIFICATION	PROFILES (CODDING)	PARTITIONS SELECTED
(DEGREE < 13) AND (AREA = "BOURGOGNE")	12, 22	a, b, d
DEGREE = 12	2.	c, d, e, f
AREA = "BOURGOGNE"	.2	a, b, d
(DEGREE = 11) AND (AREA = "BORDEAUX")	11	a, b

Figure 14. : Example of filtering of the directory

5. THE FILTERING OF PARTITIONS

5.1. Filter design principles :

The retrieval of tuples inside a partition is performed by a specialized filter performing data recognition using a finite state automaton. The time to perform any selection is then a linear function of the number of partition to filter. However, in the current version of SABRE, filters are not hardware designed : they are standard micro-processors (68 000) running a filtering algorithm that we now present.

A command received by a filter is composed of a selection expression, a list of attribute identifiers to project on and the physical address(es) of the partition(s) to filter. A selection expression is of the form :

(P11 AND P12 ... AND P1k1) OR (P21 AND P22 ... AND P2k2)
OR ... (Pn1 AND Pn2 ... AND Pnkn)

where Pij is a monomial predicate of the form :

< ATTRIBUTE-ID > . < OPERATOR > . < VALUE. >

The operator is chosen among $\{<, \leqslant, \geqslant, >, =, \neq\}$. Also, specific operators for text attributes and signature are allowed as we shall see latter. The attribute identifier is the relative number of the attribute in the tuples.

On disks, tuples are stored in fixed size partitions. Attributes are of variable length. Therefore, all attributes in a tuple are preceded by their length on one byte. A tuple is itself headed by its length on two bytes.

When executing a selection request, a filter must at first initiate the disk arm motion to the correct track. During the arm movement, the data structures necessary for filtering the data are prepared. These data structures include an automaton for each attribute appearing in the selection expression associated to a set of bit vectors. The bit vectors characterize the possible influence of the attribute on the selection expression. We are going to present more in details these data structures in the sequel.

After reading the partition (or during this time with a hardware filter), the filter scans the selected attributes for each tuple and runs the automatons. The result of an automaton execution is a bit vector. All bit vectors are then intersected : if the result is not null, the tuple satisfies the selection expression. The projection attributes are then moved to the result buffer which is the current page of the result relation (target-id relation in the SELECT command of MAPS). Whenever the bit vector becomes null or after moving all projection attributes to the result buffer, the filter examines the next tuple of the partition up to the partition end. In summary, the filtering is performed very efficiently with only two memory accesses for the selected attributes, as we are going to see in the sequel where we describe in more details the filtering preparation and the filtering execution.

5.2. Filtering preparation :

For each qualified attribute A, the influence of possible values of A on each cunjunction Cj of the selection expression :

C1 OR C2 ... OR Cj ... OR Ck

is characterized by a boolean value (true or false). Let a1, a2, ... ai ... an be the values of attribute A appearing in the selection expression, sorted in ascending order.
Let $a0 = 0$ be the minimum possible value of A and $an+1 = \infty$ the maximum possible value. Then for each cunjunction Cj and for each intervalle $]ai-1, ai[$, a boolean value Li,j giving the truth value of Cj brought by A when $ai-1 < A < ai$, is computed as follows :

.../...

$Li,j =$ { . true if A does not appear in Cj or if all monomials of Cj in which A appears are true for $ai-1 < A < ai$
. false otherwise, that is if A appears in Cj and if one monomial of Cj of type "A.operator.value" is false when $ai-1 < A < ai$

Also, for each cunjunction Cj and for each value ai ($1 \leqslant i \leqslant n$), a boolean value Ei,j giving the truth value of Cj brought by A when A = ai is computed as follows :

$Ei,j =$ { . true if A does not appear in Cj or if all monomials of Cj in which A appears are true for A = ai
. false otherwise, that is if A appears in Cj and if one monomial of Cj of type "A operator value" is false when A = ai.

In summary, to each intervalle]ai-1, ai[and to each value ai is assigned a bit (0 = false, 1 = true) for each conjunction Cj, expressing the fact that a value of attribute A in the considered range will render the expression Cj false (0) or true (1). For a given selection expression C = C1 OR C2 ... OR Ck, we obtain for each attribute A of the expression (2n+1)line vectors of k bits denoted Bi, where n is the number of values of A appearing in C.

As an example, let us consider the following expression which is a selection qualification for the WINE relation :

((VINTAGE ≠ "CHABLIS") AND (VINTAGE ≠ "CHENAS"))
OR ((VINTAGE ≠ "JULIENAS") AND (DEGREE > 12).

The bit vectors associated to the attribute VINTAGE and those associated to the attribute DEGREE are portrayed figure 15 (1 means true, 0 false).

VINTAGE	Attribute value	Cunjunction 1	Cunjunction 2	Bit Vector number
	]0, CHABLIS [	1	1	1
	CHABLIS	0	1	2
	]CHABLIS, CHENAS [	1	1	3
	CHENAS	0	1	4
	]CHENAS,JULIENAS [	1	1	5
	JULIENAS	1	0	6
	]JULIENAS,∞ [	1	1	7

DEGREE	Attribute value	Cunjunction 1	Cunjunction 2	Bit Vector number
	]0, 12 [	1	0	1
	12	1	0	2
	]12, ∞[	1	1	3

Figure 15. : Bit vectors for an example expression

Filtering preparation could stop after constructing the bit vectors tables as indicated above. For each tuple, filtering would then consist in locating the attribute values, say a, in the range a0, a1 ... an, an+1. A quick sort could be used for that. From the rank of attribute a in the sequence a0, a1, a2 ... an+1, we would get the bit vector in the table. However, with such an approach, the filtering time would be of magnitude order log2n + 2m, where m is the size of attribute A.

To reduce filtering time for each attribute to (at worst) 2m, the filter built a finite state automaton for each attribute [ROBE78]. This automaton gives at execution end the bit vector number to select in the table of bit vectors of the attribute. The automaton is just a transition matrix whose line number corresponds to the current state and column number to the input character. An element is either the next state (if positive) or the bit vector number (if negative). To reduce the size of the automaton, codding of characters in 2 bytes of 4 bits is utilized.

5.3. Filtering execution :

The filtering may be executed in parallel to the disk transfer. Tuples are read in an input buffer. As soon as one is available, the relevant attributes are checked using the automatons. An automaton delivers a bit vector Bi. All bit vectors are intersected together, starting with an initial bit register B with all bits set to true. Therefore, each time an attribute has been evaluated, all bits corresponding to cunjunctions with value false are set to 0 in register B by the operation :

B := B and Bi

Thus, whenever B is null, there is no chance that the tuple satisfies the expression ; the filter can leave out the tuple and evaluate the next one. When all attributes appearing in the selection expression have been evaluated without clearing B, then the tuple is selected.

When a tuple is selected, a second analysis is done to get the projection attributes and move them in the output buffer.

5.4. Filtering of signatures :

The filtering technique is also applied to signatures. Fragment signatures in the directory are selected by comparison with profiles. A profile is coded by a mask and a value. A mask is a bit string where a bit "1" means "significant" while a bit "0" means not. The unknown value "." is therefore coded by "0" in the mask. A profil value is a bit string where the significance of a bit is determined by the corresponding bit in the mask. Unsignificant bits are set to "0". As an example, the profile ..10 is coded as :
- mask = 0 0 1 1
- value = 0 0 1 0

Searching on a profile is done by first applying the mask (logical and) to each signature read in the directory, giving a masked signature S. Filtering is then applied to S.

6. CONCLUSION

This paper has described the design and implementation of MAPS, the content addressable page manager of the SABRE database computer. MAPS relies on three basic ideas for enhancing performance :
- a parametrized and powerful access method based on clustered predicate trees, allowing for various definitions of hashing functions,
- the intensive use of filter processors for selecting data base values as well as for selecting directory entries,
- the distribution of data among several disk units, each associated with a filter processor.

These ideas largely contribute to performance enhancement. The PT structure permits to specify and implement in a unified way various dynamic hashing schemes, such as extendible hashing, multi-attribute virtual hashing, trie-hashing and many others. Using a special purpose filter to select both entries in the directory and tuples in large partitions implies the ability to find a tuple in no more than two disk accesses. Having (with a unique implementation) a large choice of clustering strategies allow the data base administrator to place tuples satisfying most frequent queries in the same physical partitions. Finally, using one filter processor per disk increases parallelism when accessing data and thus improve response time.

7. REFERENCES

[ASTR76] M. ASTRAHAN & al. : "System-R : a relational approach to database management", ACM-TODS, Vol.1, N°2, June 1976, pp. 97-137.

[BANC80] F. BANCILHON, M. SCHOLL : "Design of a back-end for a database machine", ACM-SIGMOD Int. Conf., Santa Monica, California, May 1980.

[BANE78] J. BANERJEE, D.K. HSIAO, R.I. BAUM : "Concepts and capabilities of a database computer", ACM-TODS, Vol.3, N°4, Dec. 1978, pp. 347-384.

[BAYE72] R. BAYER, E. Mc CREIGHT : "Organization and maintenance of large ordered indexes", Acta Informatica 1, 3, 1972, pp. 173-189.

[BURK83] W. BURKHARD : "Interpolation-based index maintenance", 2nd ACM Symp. on PODS, Atlanta (Georgia), March 1983.

[CHAM81] D. CHAMBERLIN, A. GILBERT, R. YOST : "A history of System-R and SQL/Data system", 7th Int. Conf. on VLDB, Cannes (France), Sept. 1981.

[DEWI81] H. BORAL, D. DEWITT, W. WILKINSON : "Performance evaluation of associative disk designs", 6th Workshop on Architecture for non Numeric Processing, Hyères (France), June 1981.

[DEWI83] H. BORAL, D. DEWITT, D. FRIEDLAND, W. WILKINSON : "Parallel algorithms for the execution of relational operations", ACM-TODS.

[FAGI79] R. FAGIN, J. NIEVERGELT, N. PIPPENGER, H. STRONG : "Extendible hashing : a fast access method for dynamic files", ACM-TODS, N°4, 1979.

[FING81] U. FINGER, G. MEDIGUE : "Architectures multi-microprocesseurs et disponibilité : la SM90", L'Echo des Recherches, N°105, July 1981.

[GARD83] G. GARDARIN, P. BERNADAT, N. TEMMERMAN, P. VALDURIEZ, Y. VIEMONT : "Design of a multiprocessor relational database system", IFIP, 9th World Congress, Paris, Sept. 1983.

[GRAY81] J.N. GRAY : "The transaction concept : virtues and limitations", 7th VLDB Int. Conf., Cannes, IEEE Ed., Sept. 1981, pp. 144-154.

[HASK83] R. HASKIN, L. HOLLAAR : "Operational characteristics of a hardware-based pattern matcher", ACM-TODS, Vol.8, N°1, March 1983.

[HSIA83] D.K. HSIAO : "Advanced database machine architecture", Book, Prentice Hall Inc., Englewood Cliffs, New Jersey, 1983.

[KNOT71] G.D. KNOTT : "Expandable open addressing hash table storage and retrieval", Proc. ACM-SIGFIDET, Workshop on Data Description, Access and Control, 1971, pp. 186-206.

[LARS78] P. LARSON : "Dynamic hashing", BIT 18, 1978.

[LITW78] W. LITWIN : "Virtual hashing : a dynamically changing hashing", 4th Int. Conf. on VLDB, Berlin, Sept. 1978.

[LITW80] W. LITWIN : "Linear hashing : a new tool for file and table addressing", 6th Int. Conf. on VLDB, Montreal, October 1980.

[LITW81] W. LITWIN : "Trie hashing", ACM-SIGMOD Int. Conf., Ann Harbour (Michigan, USA), Sept. 1981.

[LOME83] D. LOMET : "A high performance, Universal Key Associative Access Method", ACM-SIGMOD, Int. Conf. on Management of Data, San Jose, May 1983.

[NIVE81] J. NIVERGELT, H. HINTERBERGER, K. SEVCIK : "The grid file : an adaptable symmetric multi-key file structure", Proc. of 3rd ECI Conf., 1981.

[OUKS83] M. OUKSEL, P. SCHEUERMAN : "Storage mappings for multidimensional linear dynamic hashing", 2nd ACM Symp. on PODS, Atlanta (Georgia), March 1983.

[ROBE78] D. ROBERTS : "A specialized computer architecture for text retrieval", 4th Workshop on Computer Architecture for Non Numeric, ACM, IEEE, 1978.

[ROHM81] J. ROHMER : "Applications du filtrage sequentiel", Thèse d'Etat, Grenoble, France, 1981.

[ROOM82] W.D. ROOME : "The intelligent store : a content-addressable page manager", Bell System Technical Journal, V.61, N.9, Part 2, pp. 2567-2596, Nov. 1982.

[STON76] M. STONEBRAKER, E. WONG, P. KREPS : "The design and implementation of INGRES", ACM-TODS, Vol.1, N°3, Sept. 1976.

[STON83] M. STONEBRAKER, J. WOODFILL, J. RANSTROM, M. MURPHY, M. MEYER, E. ALLMAN : "Performance enhancements to a relational database system", ACM-TODS, Vol.8, N°2, June 1983.

[SU 75] G.J. LIPOVSKI, S. SU : "CASSM : a cellular system for very large data bases", 1st Very Large Databases Conf., Sept. 1975.

[VALD84] P. VALDURIEZ, G. GARDARIN : "Join and semi-join algorithms for a multiprocessor database machine", ACM-TODS, Vol.9, N°1, March 1984.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique