INRIA
NTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

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

Rapports de Recherche

N° 203

LES ARBRES DE PRÉDICATS

Georges GARDARIN
Patrick VALDURIEZ
Yann VIEMONT

Avril 1983

LES ARBRES DE PREDICATS

G. GARDARIN, P. VALDURIEZ, Y. VIEMONT

INRIA et Université PARIS VI (Laboratoire SML)
INRIA, BP.105, 78153 LE CHESNAY-Cédex

MARS 1983

PAPIER RECUPERÉ ET RECYCLÉ

RESUME

Les index classiques par arbres balancés sont peu adaptés aux bases de données relationnelles. Nous proposons ici une nouvelle structure de données basées sur des arbres de prédicats utilisés pour regrouper les tuples d'une relation. Ces regroupements multi-dimensionnels devraient permettre d'accélérer les recherches pour un nombre important de questions.

Un des avantages essentiels des arbres de prédicats est aussi la possibilité de définir des adresses hiérarchiques, basées sur le contenu, appelées signatures. Il devient alors possible de générer des profils de signatures (signatures avec des parties inconnues) correspondant à un critère, puis, de rechercher dans des catalogues au flot des données par filtrage, les adresses physiques des partitions susceptibles de contenir des tuples répondant au critère.

MOTS CLES :

Bases de données relationnelles, chemin d'accès, recherche multi-dimensionnelle, index, prédicats, filtrage.

.../...

PREDICATE TREES

G. Gardarin, P. Valduriez, Y. Viemont

SABRE Project
INRIA
and University of PARIS VI
78150 LE CHESNAY

Abstract :

With the advent of relational database systems, multi-key searching problems have became the focus of a great deal of research. In this paper, we present a new data structure based on predicate trees for clustering tuples of a relation in a way that allows the system to accelerate a large number of multi-dimensional queries. The directories used for the implementation of predicate trees in SABRE are organized as relations which are searched efficiently by filters. One of the most significant advantages of predicate trees is the possibility of defining logical addresses based on content, called signatures, and to use filters to manage directories.

Key-words : Database machine, relational, access paths, multi-dimensional searching, indexing, predicate tree, clustering, filtering.

1. INTRODUCTION

The query optimization methods used in relational DBMS usually try to take advantage of the existing access paths to the data. These paths facilitate rapid searching of tuples having common caracteristics (e.g. some attribute value) and allow the system to replace a sequential scan of a relation by "direct" accesses to the needed tuples. Generally, the value of the access methods determines the performances of a conventional system (without specialized hardware).

The principal methods for unique key searching are hashing and B-trees [BAYE72]. Recently, new hashing schemes [FAGI79, LARS80, LITW80, SCHO81] have been proposed to handle dynamic files (files whose volume may vary rapidly) by using an index to the file (except in [LITW80]). Although very cost-effective and suitable for file systems, these methods do not fulfill the requirements of relational database systems where data are queried by predicates. Therefore, multi-key searching methods have been investigated as an attempt towards a better solution to performance enhancement of database systems. A survey of several methods can be found in [BENT79]. Several data structures such as k-d-trees [BENT75] are based on the same idea of generalizing the one-key binary search tree structure [KNUT73] to k keys. Instead of splitting overflowing nodes of the tree according to the mediane value of the same key, one of k keys has to be chosen as a discriminator at each internal node. These structures are appropriate for static files but insertions and deletions in the structure are uneasy and cause reorganisations due to the dependence between the discriminant key and the node. Recently, a dynamic file structure has been proposed in [NIEV81]. Basically, this is a way of compressing a k-dimensional bitmap for a file of k attributes. However, it is not possible to

favour the access to certain attributes rather than to others by defining a hierarchy of these k attributes. Another structure, the Reduced Cover Tree , [KARL81], was first explored in the SABRE project [GARD82]. This structure permits the implementation of multi-key clustering of data, by grouping in the same classes records (or tuples) having common key (or attribute) values. A hierarchy of n attributes defines levels of a tree and each level i is described by the alphabet of attribute i, that is, a desired set of classes chosen among the set of possibles values. However, the static definition of the alphabet, by enumeration or with a function (hashing or intervals), requires knowledge of the database evolution. Other limitations were discovered during the first implementation step, such as directory management problems.

In this paper, an enhanced method avoiding these drawbacks is described. This method is based on predicate trees. A predicate tree is associated with a relation and is dynamically defined on request of a database administrator (DBA). A list of contradictory predicates is given by the DBA for accelerating queries including such qualifying predicates. This list is used by the system to generate a level of the predicate tree. Subsequent definitions of predicate lists lead to the generation of other levels of the tree in a hierarchic way. Using a predicate tree for placing a relation is done by dividing the relation in sub-relations , each sub-relation containing the tuples satisfying a set of predicates. Of course, some sub-relations can become empty or very small. To avoid this drawback, we only divide sub-relations having a certain number of tuples. Therefore, after several levels of subdivision, we obtain a tree where certain nodes correspond to sub-relations. The predicate tree is used to accelerate all queries

including a predicate contradicting one predicate of the tree.

This paper develops the definition and the use of predicate trees for primary placement and searching in the SABRE project. It is organized as follows. The predicate tree structure and its properties are defined in section 2. In section 3, the clustering of relations using predicate trees is shown to lead to an unbalanced clustered predicate tree. The utility of predicate trees for retrieving tuples is shown in section 4.

2. PREDICATE TREES AND SIGNATURES

2.1 Definitions

Let us consider a hierarchy of p sets {p1, p2, ... pp}, each set pi of mi monorelation contradictory predicates Pi1, Pi2, ..., Pimi. A predicate Pij is a set of atomic predicates expressed in disjunctive normal form. Each atomic predicate is of the form R.att θ value, where R.att is an attribute of relation R and θ an operator chosen among $\{=, \leq, \geq, <, >\}$. A predicate Pij can be seen as a restriction predicate over relation R. The union of the mi sub-relations resulting from restriction operations by predicates Pi1,Pi2,...Pimi would give relation R.

A predicate tree (PT) of depth p is a balanced tree where each level i is defined by the ith set of predicates pi, each node of level i-1 having mi sons corresponding to predicates Pi1, Pi2, ... Pimi. The number mi of predicates in a set i is called the branching factor. A path from the root to a leaf corresponds to the logical intersection of all the predicates located in the nodes of this path. We say that a predicate tree is consistent if each path from the root to a leaf corresponds to a predicate which can be satisfied by at least one tuple.

In the sequel, we only consider consistent predicate trees.

Allowing a hierarchy of the p sets of predicates is an important feature of PTs. It permits the assignment of a decreasing priority to the sets of predicates. The predicates are more favoured as they are close to the root of the tree. For example, let us consider two sets {P11, P12, P13, P14} and {P21, P22}. The resulting PT is portrayed figure 1.

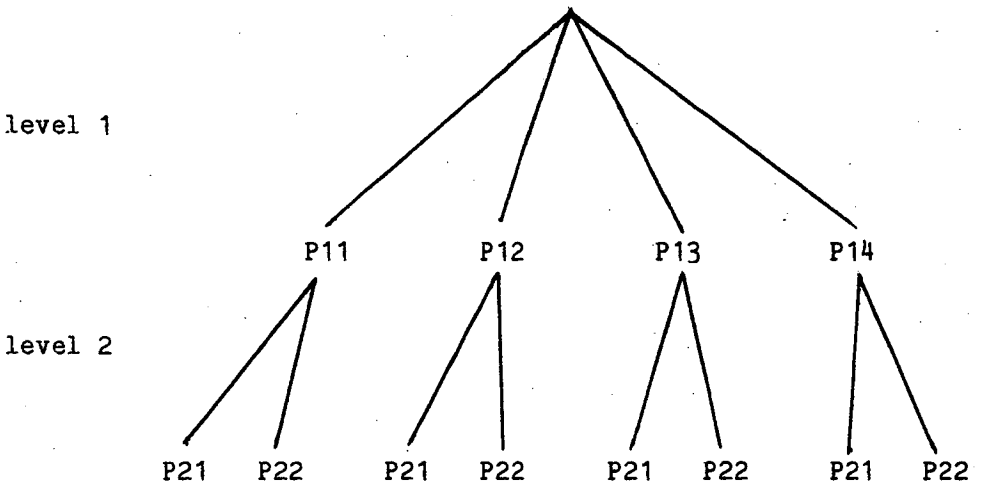


Figure 1 : First example of a predicate tree.

The access to predicate P12 is done by locating only one node. The access to predicate P22 needs to find four nodes (one for each predicate of level 1). However, the access to predicate (P12 and P22) corresponds to only one node. In order to provide a fully dynamic structure, it is desirable to allow additions and deletions in the tree. The simplest addition corresponds to the definition of a new level p+1 in a tree of depth p. A deletion means a suppression of a level i of the tree and is a more complex operation.

As a more meaningfull example, let us consider the relation CUSTOMER (CUST#, NAME, CITY, DEB, CRED) and the following two sets of countradictory predicates expressed in a tuple relational calculus like language.

p1 = (CITY=PARIS and CRED<100; CITY=PARIS and CRED≥100;

CITY=CANNES or CITY=NICE)

p2 = (CUST#<2000; CUST#≥2000 and CUST#<3000; ... ; CUST#>10000)

Note that in p1, the credit of customers living in Cannes or Nice is not useful for selections. The predicate tree defined by {p1, p2} is depicted in figure 2. The branch "OTHERS" is generated if necessary as the complement predicate of the mi specified predicates. It is useful for p1 to qualify customers who do not live in PARIS, NICE or CANNES. The encircled predicate corresponds to the restriction predicate permitting to obtain the customers such as CITY=PARIS and CRED≥100 and CUST#≥2000 and CUST#<3000.

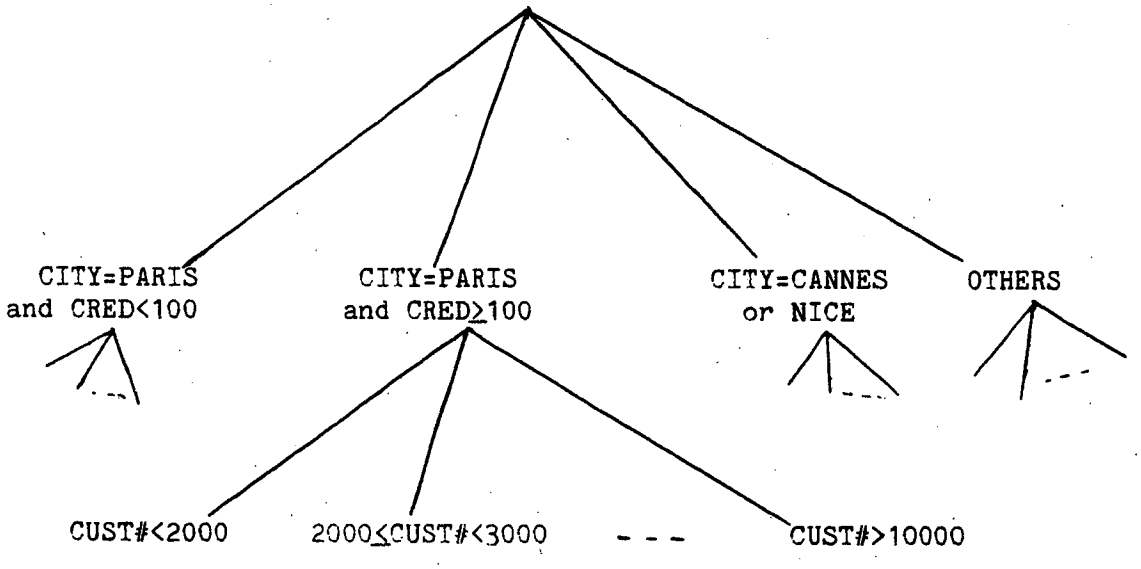


Figure 2 : a Predicate Tree for the CUSTOMER Relation

2.2 Addressing of nodes in a PT

To address a node in a PT, we first label each external arc of a node by the rank of the predicate in the list of predicates associated to the level. Then we utilize the path from the root to address each node. Such an address is called a "signature". More precisely, a signature of a node is the list of arc labels from the root to the node.

In the SABRE system, a signature of a node of level i is implemented as a bit string obtained by the concatenation of a serie of i integers, where each of them is the predicate number of level j in the path from the root to this node. Thus a signature is a logical address which uniquely identifies a node and in particular a leaf in a tree. The most important property of a signature is that it depends only locally on the structure of the tree, i.e. not affected by a modification elsewhere in the tree except one involving the path from the root to the corresponding node. This property will be used in the following. Figure 3 illustrates a predicate tree of depth p=2 and the signatures of the encicled leaves. For the purpose of clarity, only two nodes of level 1 are developed.

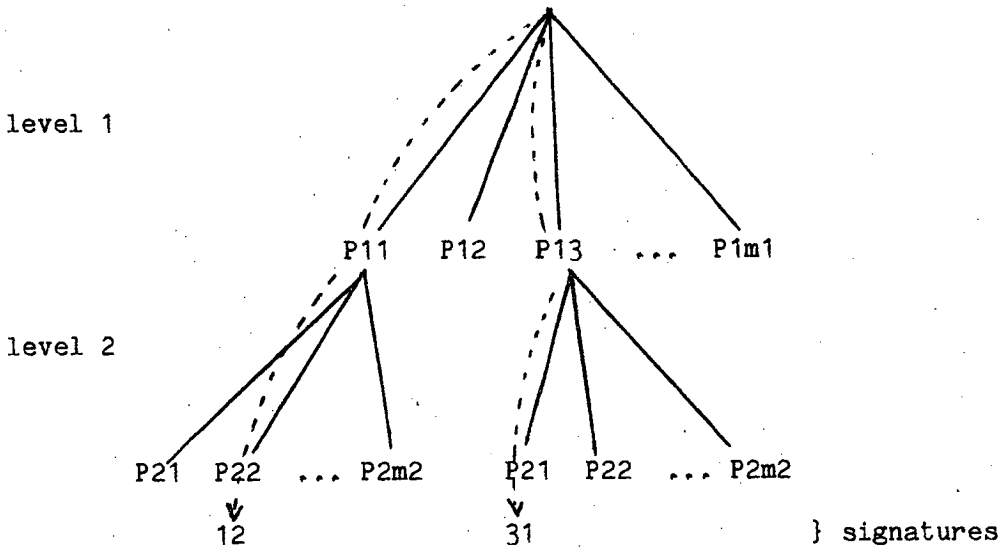


Figure 3 : A Predicate Tree and 2 leaf Signatures

A useful manipulation in a predicate tree is the addressing of several nodes having common characteristics. For instance, a subtree or the same leaves in several subtrees correspond to some particular restriction predicates. An obvious way for addressing nodes in a PT is to provide the list of signatures. However, this list can be very large

for a large number of predicates. To address subtrees, we introduce the notion of signature profile : It is a signature where some digits remain unspecified, meaning "0" or "1". Obviously, each digit of a profile is coded with two bits, in order to enable three values: "0", "1" and undefined noted ".". The addres of a subtree, considered as the set of its nodes, is therefore given by a signature profile, composed of a signature prefix of lenght q, pointing a subtree of depth p-q. An example of such a signature profile is given figure 4, where p=2 and q=1. The selected nodes are marked. We assume that each level corresponds to two digits in a signature profile among {0,1,.}.

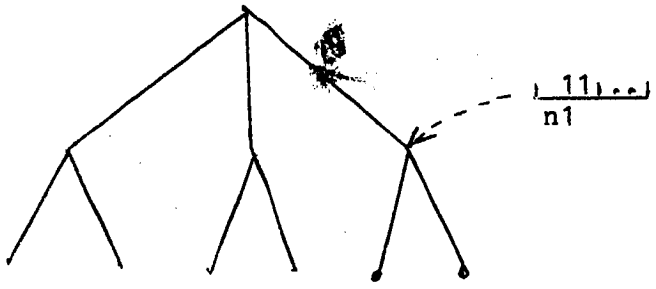


Figure 4 : A Signature Profile and the selected leaves.

More generally, the unspecified value can appear anywhere in a signature profile. An example of such a signature profile is illustrated figure 5, where p = 3. Such a profile addresses several subtrees.

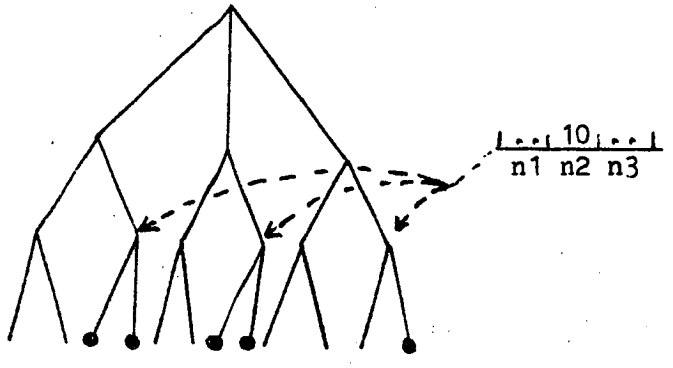


Figure 5 : A general Signature Profile.

3. CLUSTERING USING A PREDICATE TREE

3.1. Clustered Predicate Tree

Now, the use of predicate tree to cluster tuples in a partitioned memory is considered. We assume that the storage allocation of a relation on disk is done by units of fixed size called partitions. A partition can be read in one access and is supposed to hold several tuples.

A predicate tree as defined in section 2.1 can be utilized to cluster tuples by assigning a cluster to each leaf. Each tuple in the cluster satisfies the intersection of all the predicates on the path from the root to the leaf corresponding to that cluster. As we defined the predicate tree as a balanced tree where each level is developed, assigning a cluster 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 cluster. If all the sons of a father are grouped in a single cluster, then the father becomes itself a leaf. This process of grouping underoccupied clusters can occur until obtaining an almost full cluster, according to a 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 clusters with nodes of a predicate tree is called a clustered predicate tree. We can define more formally 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 1 is illustrated figure 6.

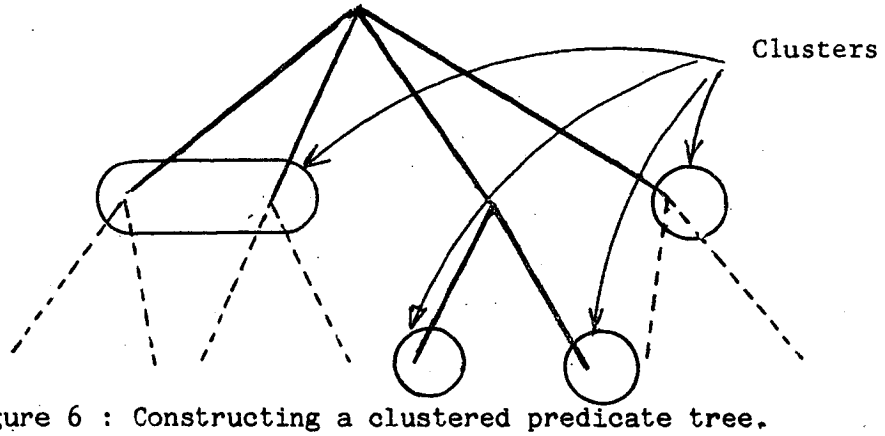


Figure 6 : Constructing a clustered predicate tree.

3.2 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 problem for finding the partition address where a tuple must be inserted will be addressed in the next section. Let us assume first that the relation is empty. Then the first tuples are inserted in a single partition previously allocated to the relation. A problem arises when the partition overflows. If the order q of the clustered predicate tree is greater than 1, then a new partition is just allocated to the cluster. The real problem arises when q partitions are full, that is, when the root cluster is saturated.

More generally, when a cluster of level (i-1) is full, it is split according to the mi predicates of the i-th level of the predicate tree. Therefore, the clustered predicate tree grows by addition of mi branches to the node corresponding to the saturated cluster. Each branch j defines a new cluster which contains tuples satisfying the j-th predicate of level i, that is Pij. On a physical point of view, if at

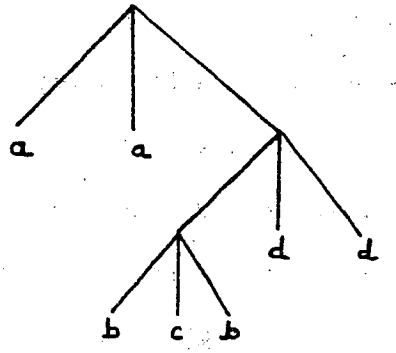
least one partition was allocated to each cluster, the average partition occupancy could be very low, each having approximately q/mi bytes occupied. To avoid this waste of space, several clusters are placed in the same physical partitions. Therefore, a saturated cluster is logically split in mi clusters, but physically split in two parts of approximately q/2 partitions. The next splittings of these partitions will be made horizontally with regard to the predicate tree, that is , by redistributing the clusters in partitions until each physical partition contains only tuples of one cluster. Then when a cluster overflows (i.e. is alone in more than q partitions), it is vertically split according to level (i+1) if it exists in the predicate tree. Consequently, the storage occupation of partitions stays greater than 50%. When a cluster overflows at the bottom level of the predicate tree (i.e., the corresponding node in the predicate tree is a leaf), it cannot be split. In this case, we simply add a new partition to the cluster. That leads to terminal clusters with more than q partitions.

The reverse process is the deletion of tuples in a relation. First,when the occupancy factor of a partition becomes lower than a given value (for example 40%), we try to merge it with a contiguous brother partition such that the sum of their occupancy factor is less than 1. This corresponds to horizontal splitting, that is, to the fact that we try to store several clusters in a unique partition. Then, when the sum of all occupancy factors of all the sons of the father is less than q, we group all these sons in a unique cluster. This corresponds to the definition of a clustered predicate tree, which shrinks as clusters disappear.

3.3. Representation of a clustered predicate tree

A clustered predicate tree describes the way a relation is clustered. It is then necessary to memorize the structure of the tree and the physical address of partitions containing the clusters. A signature of length s , as defined in section 2.2 , stands for a logical address to a node of level s in the clustered predicate tree. A method to represent a clustered predicate tree is to keep the logical-physical mapping. We store it in a relation with three attributes <relation id, signature, partition address>, called a directory. Using a relation for the directory permits to apply the same tools as for the other relations . In particular, we can utilize filters [BANC81] to scan the directory; it is a most attractive device to manage clustering via predicate trees , as we show in the discussion that follows. Also the directory provides a substantial logical-physical independence .

A representation of a clustered predicate tree is portrayed figure 7, with its corresponding subset of the directory. The PT associated with the relation is a balanced tree with 3 predicates by level giving 27 leaves.



rel.id	signature	address
R	1	a
R	2	a
R	311	b
R	312	c
R	313	b
R	32	d
R	33	d

Figure 7 : a Clustered Predicate Tree for a relation R

3.4. Clustering algorithm in SABRE

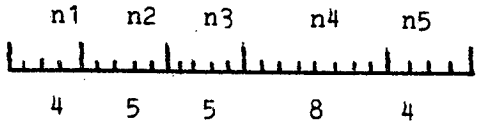
In the SABRE database machine, the placement of a relation is described by a clustered predicate tree of order 1, described in the

directory. The directory is searched by filters which are extended to handle signature comparisons.

The signature, regarded as an attribute, must be as compact as possible to limit the size of the directory and must permit easy selections. Each signature part ni corresponding to a level of the tree is bounded by mi, the branching factor of level i of the tree. Therefore, each part ni can be coded as a number bi of bits in which the corresponding maximum mi can hold. The number bi is defined by :

$$2^{**}(bi-1) < mi \leq 2^{**}bi$$

For example, the signature of length 5 with the branching factors 16, 20, 20, 200, 10 is coded by the bit string:



This string needs only 4 bytes and 12,800,000 signatures can be generated. The bit string is an attribute type of SABRE. We use the zero length attribute (null value) as a signature for the root.

To select sub-trees corresponding to a given node in the clustered predicate tree, we introduce a new comparator suitable for selecting on the signature attribute type in the directory. This comparator , noted <>, enables us to test the prefix relation between two signatures. Two signatures are in relation prefix if and only if one of them is a prefix of or equal to the other. In particular, the zero length signature is a prefix for every other one. For example, we have :

010 <> 010 ; 010 <> 01011 ; 01011 <> 010

but we have not: 011 <> 01011 . Used with a signature profile as an argument of a selection, this comparator allows the system to scan the directory in order to retrieve all the relevant signatures for a given

query, and then scan the corresponding partitions. The applied rule is that every "." value (undefined value) in the profile matches both "0" and "1" in the signatures.

The algorithm for finding the partition address where to insert a new tuple in a relation is rather simple. It can be summarized as follows :

1. With the PT associated With the relation, generate a signature S for a tuple (by testing predicates on tuple attributes).
2. Select in the directory the entries having a signature in relation prefix with S.
3. If a unique entry is found, it gives the partition address where to place the tuple.

   If several entries are found, the last allocated partition is chosen to insert the tuple.

   If no entry is found, this is the insertion of the first tuple in the relation; therefore, a partition is allocated and an entry in the directory is created with a zero length signature.

The first step does not require any physical knowledge and uses the predicate tree. The second step can be implemented by a filter performing the prefix relation selection on the directory. The tuple is then placed in the partition. The case of overflow, discussed in section 3.2, leads to the creation of entries in the directory with signatures longer than the selected one and the deletion of the selected entry. Only one new partition is then allocated.

3.5 Organization of the directory

The size of the directory is proportionnal to the size of the addressed relations. A directory of a single 4K byte partition could

address about 500 4K byte partitions. Accordingly, the directory is placed itself using a PT . The first level of the PT is defined using the relation id which is an integer. Each predicate Pi1 is modulo(rel-id,p) = i, where rel-id is the relation identifier and p a generation parameter. Thus, a directory partition can hold addresses of different relations. When a relation becomes big enough, one directory partition is not sufficient , so new levels of the directory predicate tree are considered. As the signature contains information about the content of the physical partition, it is used for the clustering of the directory. Thus, successive groups of 4 bits of the signature define other successive levels, each having 16 branches. Therefore, a leaf in the placement tree of the directory corresponds to a subtree in the placement tree of the relation.

4. SELECTION ALGORITHM

4.1 The refutation of clusters

The predicate tree and the associated clustered predicate tree are used to translate a mono-relation query qualification (restriction type query) into an equivalent query against a smaller number of clusters, that is to refute clusters. Let Q be the query qualification. The refutation algorithm generates cluster addresses which can contain tuples satisfying the query qualification. The principle of the algorithm is to determine the predicates contradictory with Q. Clusters corresponding to those predicates are eliminated from the list of possible clusters. To address clusters, we utilize signature profiles. Therefore, the refutation algorithm generates a list of signature profiles which determine the clusters containing candidate tuples for

answering the query. The relevant signature profile parts generated when examining level i of the predicate tree are in fact the branch numbers which corresponds to not refuted clusters. To generate these branches numbers, two cases can be considered. First, if the set of predicates Pi of level i in the tree has no common attribute with the qualification Q, then no branch can be refuted and the unknown signature profile part is generated. (Recall that unknown which is denoted ..... match with everthing.) Otherwise, if Pi has common attributes with Q, then for each predicate Pij which is not contradictory with Q, j must be kept as a possible signature profile part. Finally, by combining all signature profile parts, we obtain a list of signature profiles which determines clusters to scan in the relation. Note that, at any level, an excessively long list of signature profiles can be replaced by a shorter one by substituting unknown to the signature part of the level.

4.2 The search algorithm

The search algorithm is composed of two steps. In the first step, the directory is scanned to search the relevant partition addresses. More specifically, the signatures in relation prefix with the signature profile list generated by the refutation algorithm are selected in the directory. All physical partition addresses corresponding to those signatures are kept. This first step allows the system to find physical addresses of all partitions which can contain candidate tuples for answering the query.

The second step consists in scanning the partitions whose addresses were retrieved in the first step. The query (or a simplified query) is applied to those partitions to get the response tuples. In summary, the answer to a query is constructed by two successive fiterings. The first

one is applied to the directory ; the second one is applied to the relevant partitions.

4.3 Detailled algorithms with mono-attribute predicates

We now describe the selection algorithm in the case of a PT where , for the sake of simplicity, is defined on only one attribute. This kind of PT permits to define intervals or hashing functions at each level. We assume here that the query is defined as:
{X / Q (X)} where X is a tuple variable and Q a qualification expressed in disjunctive normal form. Thus, we have :

$$Q = (Q11 \wedge Q12 \wedge .. \wedge Q1k1) \vee ... \vee (Qn1 \wedge ... \wedge Qnkn)$$

where Qij has the form X.A θ value, A being an attribute of the queried relation, θ a comparator. The refutation algorithm determines the signature profiles to answer the query. Then, the signature profiles are searched by the filters performing a prefix relation selection on the directory. Finally, data partitions are scanned by the filters and response tuples are selected with a simplified query.

A Boolean vector is associated with Q, where each entry is a boolean value of a simple predicate. The vector is initialized to false. The selection algorithm can be summarized as follows: (1) Project the query Q on the attributes involved in the PT; (2) Select convenient predicates in the PT and replace Q by an equivalent query Q'; (3) Generate the signature profiles to answer Q'; (4) Access the directory to find the partitions addresses corresponding to the list of signature profiles; (5) Access the partition and apply Q'. A more detailed description of the algorithm can be stated as follows.

1. Set to true the entries of the boolean vector for simple predicates on non relevant attributes and simplify Q with the simple rule:
   P ∧ true -> P; P ∨ true -> true;
2. For each remaining conjunctive sub-expression Qi
   for each level x of the tree on attributes of Qi
   - select a sub-query SQi of Qi such as there exist predicates
     Pxy : Pxy -> SQi
   if "SQi found" then
   - store the branch numbers corresponding to level x in a unique list Lx
   if SQi -> Pxy then (here: SQi and Pxy are equivalent)
   - replace Qi by Qi' where Qi = Qi' ∧ Pxy
   - replace Q by Q' = Q'1 ∨ Q'2 ∨ ... ∨ Q'n
3. For each level x of the tree where no Lx exists, make Lx initialized with null.
   With all the lists Lx, generate all signature profiles by nested loops on numbers nxj in lists Lx. (Note that signature profiles may be viewed as a binary tree).
4. Read the directory and select the partitions addresses where the signatures match the disjunction of signatures profiles according to the "<>" selection.
5. Read each data partition and select the tuples matching Q'

Step 4 can occur two times in case of a two level directory. Always, the same signature profiles are used to search the directories since the placement of the directories is given by the signatures themselves. Generally the signature profile list will be very short, but occasionally it may be long because the entire PT is consulted and the query may be very complex. The solution to simplify the selection by the filters is to use a linear semi-join algorithm [RHOM80].

5. CONCLUSION

The predicate tree structure herein proposed provides multi-key access to static as well as dynamic files. The use of predicates to define the relation organization is uniform with the data manipulation language and simplifies data base administration. A signature in a predicate tree provides an invariant logical addressing in a partitionned memory and gives information about the content of a partition. The directory used for the logical-physical mapping can be searched efficiently by using signature profiles. In the SABRE database machine, this searching is done by filter processors. The directory itself can be partitionnedd by a predicate tree deduced from the original tree.

The proposed structure enables the placement of a relation by multi-dimensional clustering or on unique key, where a directory associates primary signatures with partitions addresses. The structure also allows the system to index a relation on secondary key, with a directory associating secondary signatures with primary signatures. Finally, the selection algorithm exhibits the advantage of using Predicate Trees to fulfil the needs of queries expressed in a non procedural language, as in SABRE.

6. REFERENCES

BANC80 F. Bancilhon, M. Scholl
"Design of a Back-end Processor for a Database Machine" Proc. of the ACM-SIGMOD, Santa Monica, Ca, May 1980.

BAYE72 R. Bayer, E. McCreight
"Organization and Maintenance of Large Ordered Indexes"
Acta Informatica 1, 3 , pp 173-189, 1972.

BENT75 J. Bentley, J. Friedman
"Analysis of Range Searches in Quad Trees"
Inf. Process. Letters, 3, July 1975.

BENT79 J. Bentley, J. Friedman
"Data Structures for Range Searching"
Computing Surveys, vol. 11, no 4, December 1979.

FAGI79 R. Fagin, J. Nivergelt, N. Pippenger, H. R. Strong
"Extendible Hashing : a Fast Access Method for Dynamic Files"
ACM-TODS no 4, 1979.

GARD82 G. Gardarin, P. Bernadat, N. Temmerman, P. Valduriez, Y. Viemont
"SABRE : a Database System for a Multi-microprocessor Machine"
2nd Int. Workshop on Database Machines, San Diego, September 1982.

KARL81 K. Karlsson
"Reduced Cover-Trees and their Application in the SABRE Access Path Model"
Proc. of 7th Int. Conf. on VLDB, Cannes, September 1981.

KNUT73 D. E. Knuth
"The Art of Computer Programming, vol. 3 : Sorting and Searching"
Addison-Wesley, Reading, Mass., 1973.

LARS80 P. Larson
"Linear Hashing with Partial Extension"
Proc. of 6th Int. Conf. on VLDB, Montreal, October 1980.

LITW80 W. Litwin
"Linear Hashing : a New Tool for File and Table Addressing"
Proc. of 6th Int. Conf. on VLDB, Montreal, October 1980.

NIEV81 J. Nievergelt, H. Hinterberger, K.C. Sevcik
"The GRID FILE : an Adaptable Symmetric Multi-key File Structure"
Trends in Information Processing Systems, Proc. 3rd ECI Conf., 1981.

PFAL80 J.L. Pfaltz, W.J. Berman, E.M. Cagley
"Partial-Match Retrieval Using Indexed Descriptor Files"
Communication of the ACM, Vol 23, no 9, September 1980.

ROHM80 J. Rohmer "Machines et Langages pour traiter les ensembles de donnees (textes, tableaux, fichiers)" These d'etat, Grenoble, December 1980.

SCHO81 M. Scholl
"New File Organizations Based on Dynamic Hashing"
ACM-TODS, vol 6, no 1, 1981.

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique