MG-Tree: An Efficient Index for Fast Execution of Conjunctive Queries

Yuping Yang and Mukesh Singhal
Department of Computer and Information Science
2015 Neil Avenue
The Ohio State University
Columbus, OH 43210-1277, USA
Tel: (614) 292-5839
Fax: (614) 292-2911
{yangy,singhal}@cis.ohio-state.edu

September 29, 1998

Abstract

Complex relational queries with multiple conjunctive (AND) predicates have attracted increasing attention from database industry and research communities for their importance in decision making and online information systems. Previous attempts that use signatures to execute conjunctive queries did not separate the signatures extracted from different attributes. Such a mix of meta-data from different attributes makes implementation difficult and reduces search efficiency.

This paper introduces MG-tree, which completely eliminates the mix of meta-data from different attributes and combines the strength of both tree-like indexes and signature files to support the execution of conjunctive queries. An MG-tree is created at one attribute and its signatures are collected from other attributes. One MG-tree can replace several tree-like indexes and is much smaller than the tree-like indexes it replaces. Because signatures of several attributes co-exist, but not mixed, in each node of an MG-tree, for answering conjunctive queries, accessing data through an MG-tree can even be faster than through several tree-like indexes. A crucial distinction between our approach to previous multi-dimensional signature based index is that we construct higher level signatures by bitwise-OR the signatures column-wise, not tuple-wise.

Our analyses and experiments show that our approach achieved a significant access speed improvement over both tree-like index and previous multi-dimensional signature indexes. As a small and fast access index for executing conjunctive queries, the MG-tree can be useful in practical DBMS implementations.

Key words: Signature, access index, conjunctive query.

1 Introduction

Tree-like access indexes such as B-trees [1] or B+ trees have been used extensively in database implementation, notably in relational databases. A tabular data file which has a tree-like index such as a B+ tree at each of its columns is called a fully inverted file. A fully inverted file has very fast access speed but a very large storage space overhead. Often the index itself takes as much space as the data itself, or even more.

A tabular dataset is a set of data organized in a two dimensional matrix. To conform to the terminology in relational database research, a row and a column in a tabular dataset are called a tuple and an attribute, respectively.

Alternatively, signature file can be used as an access index and it has been studied extensively in the past, mostly in the context of text retrieval [3, 7, 9, 13, 12, 14]. Compared to tree-like indexes, signature files are much smaller access indexes. Often a signature file is only 10%-20% of the size of the primary dataset [6]. A signature is a vector of 0/1 bits hashed from a data item, such as an English word, an address, an image object, etc. Several signatures of the same length can be bitwise-ORed to generate a group signature. A bit of the group signature is "1" if at least one of the generating signatures has "1" at this bit position. For example, the bitwise-OR of "10010" and "00110" results in a group signature "10110". A group signature is a compact form of meta-data for its generating signatures.

As an example of using signatures, suppose a query is looking for a word "desktop" in a data file ("PC", "workstation"). "PC" is hashed into "10010" and "workstation" is hashed into "00110". So, the data file has a group signature "10110". If the word "desktop" is hashed into a query signature "01100", because the 2nd bit of the query signature is "1" and the corresponding bit of the group signature of the data file is "0", it is clear that the queried word cannot exist in the data file. So, signatures can be used to filter out large quantities of irrelevant information during query execution.

When a query signature matches a group signature, it may not be true that the queried data item is in the data file. As an example, suppose "10100" is hashed from "mainframe". A query that is looking for "mainframe" has its query signature "10100" match with the group signature "10110", in the sense that each bit of "1" in the query signature corresponds to a bit of "1" in the group signature. However, the word "mainframe" is not in the data file ("PC", "workstation"). This is called a false drop. The ratio of the number of false drops to the number of all drops is called the false drop ratio of the signature file. The false drop ratio can usually be made low enough by using signatures with sufficient length.

Previously, Chang and Schek [11] has described a multi-dimensional signature index to execute conjunctive queries in Starburst database system. In their approach, group (tuple) signatures are formed by bitwise-OR signatures from different attributes as shown in Fig.1. It is difficult to distinguish information from different attributes. We will go to detail about

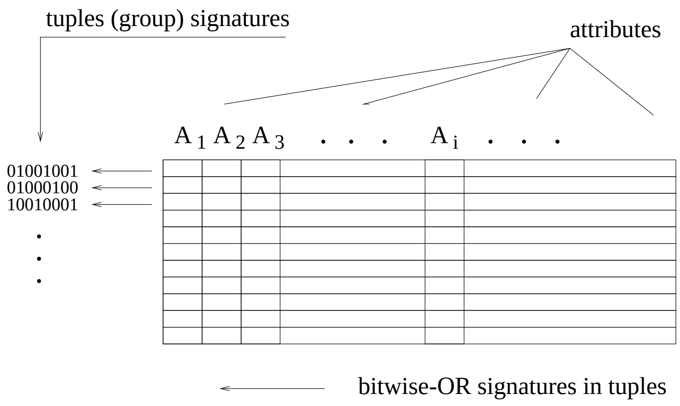

Figure 1: Construction of tuple (group) signatures.

this in Section 5.

We introduce a *multi-attribute signature enhanced tree*, MG-tree, which does not mix information from different attributes. Signatures hashed from data items in an attributes are bitwise-ORed with signatures in the same attributes. An MG-tree can be built from a B+ tree and can replace several B+ trees. The size of an MG-tree is much smaller than the size of all B+ trees it replaces. An MG-tree is especially good for answering conjunctive queries.

The crucial distinction between our approach and previous multi-dimensional signature indexing techniques is that we do not mix meta-data from different attributes.

This paper is organized as follows. Section 2 introduces the concept of the MG-tree. Section 3 analyzes the relationship between the size of an MG-tree and false drop ratios, and both analytical and experimental results are presented. Section 4 compares the access speed between an MG-tree and multiple B+ trees for answering conjunctive queries. Section 5 discusses the difference between the MG-tree and some of the related works. Finally, Section 6 summarizes the paper.

2 MG-Tree

Though an MG-tree can be built by augmenting signatures to the nodes of any tree-like index, we focus our discussion on B+ tree for its importance in real-life systems. Suppose a B+ tree is created at an attribute A_t. The leaf nodes of the B+ tree contain values in the data domain of A_t and also contain pointers pointing to the tuples of R. Each tuple in R is pointed to by a pointer from a leaf node of the B+ tree. Pointers in a (leaf or non-leaf) node are separated by values from the data domain of A_t and these values are used to guide search processes to locate appropriate pointers to nodes/tuples of lower levels which lead to true query results. An MG-tree can be built from such a B+ tree.

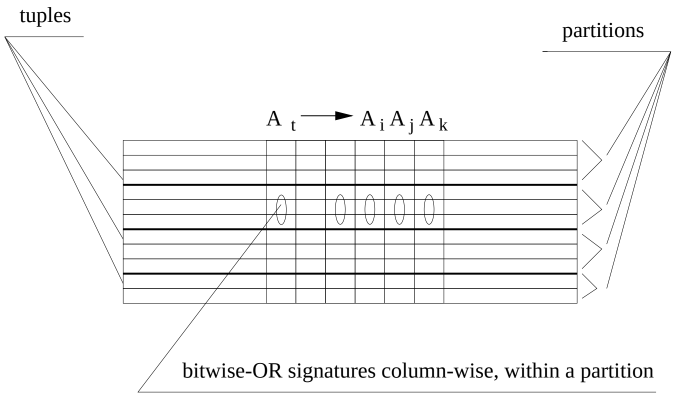

Figure 2: A partitioning of a tabular dataset.

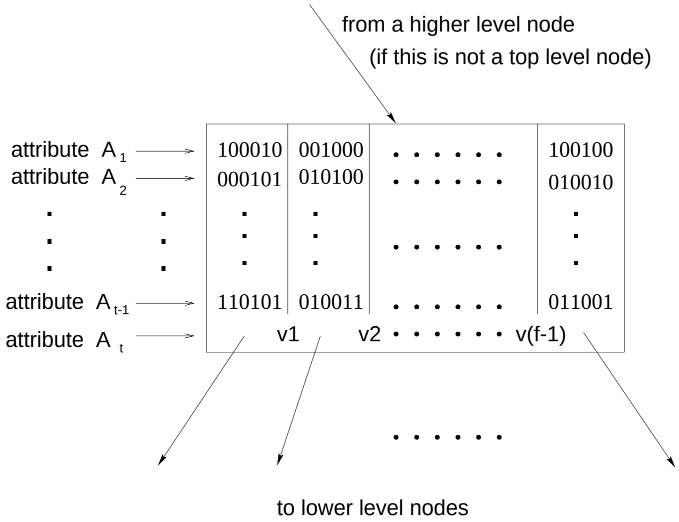

Figure 3: A typical non-leaf non-top node of an MG-tree with t attributes.

Structure of an MG-tree: Fig. 2 shows a partitioning of the tuples of R. A partition is a set of tuples. A group signature is generated from each attribute in each partition. If R has 20 attributes and is partitioned into 100 partitions, 2,000 group signatures will be generated. To make a group signature, signatures that directly hashed from data items are bitwise-ORed along each column within each partition. This is a crucial difference between our method and previous methods [10, 11].

Fig.3 shows a typical non-leaf node of an MG-tree. Pointers in the node point to lower level nodes. The MG-tree is created at attribute t and group signatures are collected from attributes A_1 through A_{t-1}. For each attribute A_i, $1 \leq i \leq t-1$, exactly one group signature is associated with each pointer in the node. For the attribute A_t, v_1, v_2, ..., v_{f-1} are values from the data domain of attribute t and are used to separate pointers pointing to lower level nodes. f is the fan out of the node. All attributes except A_t have group signatures in the

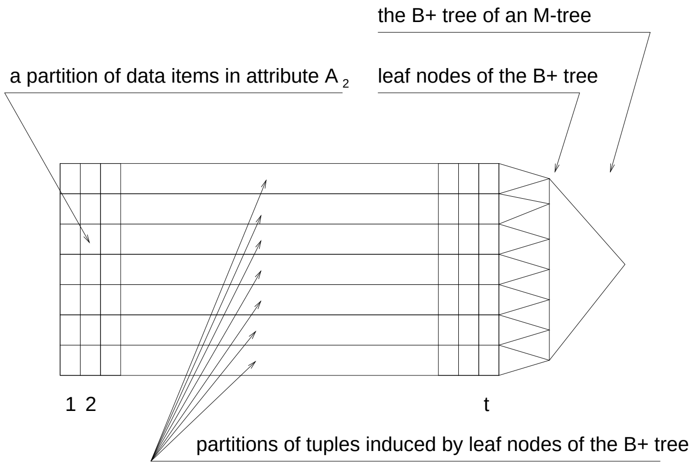

Figure 4: The partitioning of tuples by leaf nodes.

node.

A leaf node has only values and pointers directly point to tuples of R. A leaf node represents a group of tuples in R and corresponds to $t-1$ group signatures, one for each attribute A_i, $1 \leq i \leq t-1$. However, these group signatures are not stored in the leaf node and are entered in the second level non-leaf node which is the parent node of this leaf node.

Query execution starts from scanning the highest non-leaf nodes. When executing a query having predicates involving attribute A_{q_i}, $q_i < t$, only group signatures belong to attribute A_{q_i} are checked. If $q_i = t$, then values v_1, v_2, ..., v_{f-1} are also checked. The checking of group signatures and/or values v_1 through v_{f-1} determines which pointer need to be further traversed for the query execution.

We use an example to further illustrate the detailed construction of an MG-tree from a B+ tree. Assume R has 125k tuples and 20 (t=20) attributes. A B+ tree T with three levels is constructed at A_t. By definition, the fan out of each node of a B+ tree has a lower and a higher limit. For simplicity of explanation, we assume that the fan out of each node is always 50 (f=50).

Counting from the bottom up, there are 2,500 (because 2,500 $\times$ 50 = 125k) leaf nodes at the bottom level of the B+ tree, 50 (50 $\times$ 50 = 2,500) nodes at the middle level, and one node at the top level. Each node contains 49 values and 50 pointers. For a non-leaf node, the 50 pointers point to 50 lower level nodes. For a leaf node, the 50 pointers point to 50 tuples. The size of each node in T as a B+ tree is 49 s_v + 50 s_p, where s_v and s_p are sizes of a value and a pointer in bytes, respectively. The size of the B+ tree T is $(2{,}500 + 50 + 1) * (49\ s_v + 50\ s_p) \approx 13 * 10^4(s_v + s_p)$ bytes.

To make T into an MG-tree, group signatures need to be constructed. Let us focus our discussion on attribute A_2. The 2,500 leaf nodes divide the tuples of R into 2,500 partitions as shown in Fig.4. The partitions of tuples induce partitions of data items in each of the attributes A_1 through A_{19}. So, values of attribute A_2 are partitioned. Every value in a partition of A_2 contributes (by a hash function ϕ_1) a signature towards the construction of a group signature in attribute A_2. The group signature of a partition is obtained by bitwise-ORing all signatures in that partition from attribute A_2.

The constructed group signatures as described above are entered into middle level nodes in T. Each middle level node has 50 group signatures because f=50. Each group signature corresponds to one pointer pointing to a leaf node. In each of the middle level node, there are total of (because there are 19 attributes with signatures, and each attribute has 50 signatures) $50 * 19 = 950$ group signatures.

Several methods can be used to construct group signatures for attribute A_2 at the top (third) level. One way is to bitwise-OR group signatures from middle level nodes to obtain signatures at the top level [4]. Each top level group signature is bitwise-ORed from 50 middle level group signatures of an attribute.

Another way is to use a different hash function ϕ_2 to directly hash out a different signature for each value in attribute A_2 and then bitwise-OR these signatures to obtain group signatures at the top level [5, 16]. In this approach, each top level group signature is bitwise-ORed from 2,500 signatures which are hashed directly from data items of an attribute.

Below we present an algorithm for construction of an MG-tree. Assume that the number of attributes of R is t and the fan out of each node is f. A second level non-leaf node represents f^2 tuples.

1. pick an attribute A_t in R and create a B+ tree index on it.

2. the tuples of R are partitioned by leaf nodes of the B+ tree.

3. the values in each attribute are partitioned by the tuple partitions.

4. each partition in each attribute (A_i, $i \neq t$) generates one group signature.

5. each second level non-leaf node stores $(t-1)f^2$ group signatures.

6. if there are higher level non-leaf nodes, group signatures in the second level nodes are further bitwise-ORed along each attribute to generate higher level group signatures.

Note that the partitioning of tuples and values are naturally induced by leaf nodes of the B+ tree and no actual operation is performed in step 2 and 3. Also note that leaf nodes contain no signatures, though each leaf node corresponds to t group signatures. We want to emphasize that the partitioning is a process to assign tuples to the leaf node in a natural way and there is no movement/clustering of tuples to be done.

There are many ways to construct multi-level signature files in the MG-tree. The separate coding method [5, 16] provides an excellent performance. A separate coding file is costly to maintain if the number of levels is high [16]. Actually this is not a too severe restriction in practice because usually nodes in a B+ tree have large fan-outs and a B+ tree has only a few levels. Alternatively, if low maintenance cost is strongly desired, then index descriptor file [4] may be used.

Using an MG-tree to answer conjunctive queries: When accessing data in attribute A_t, the MG-tree T is used exactly as a B+ tree. When accessing data in attribute A_i, $i \neq t$, T is used as a multi-level signature file. However, T is much more than a B+ tree and a set of multi-level signature files put together. The real advantage of using an MG-tree is in answering conjunctive (AND) queries. Conjunctive queries frequently appear in decision making support systems and online service systems and have been receiving increasing attention in the research community.

Suppose a conjunctive query has three predicates: $(R.A_t = a)$ AND $(R.A_2 = b)$ AND $(R.A_3 = c)$. To answer this query, two query signatures S2 and S3 are made out of b and c, respectively. No query signature is made for a.

We use the MG-tree described above in our example. The MG-tree has three levels. At the top level node, by comparing value a with 49 values of A_t, a set P1 of qualified pointers is selected. In the top level node, each attribute other than A_t has 50 signatures and each signature corresponds to a pointer pointing to a middle level node. Signature S2 is checked with the signatures from A_2 and the pointers of matched signatures are selected into P2, which is a pointer set. Similar process can be done for S3 to obtain a qualified pointer set P3. Each pointer set represents the query execution result of one predicate at the top level. If $P1 \cap P2 \cap P3 = \emptyset$, then it is clear that the query will return an empty set. Otherwise, the search process follows pointers in $P1 \cap P2 \cap P3$ and proceeds to nodes at the middle level.

For each middle level node which is pointed to by a pointer in $P1 \cap P2 \cap P3$, the search is done similar to that at the top level node and a pointer (selected from pointers from this middle level node) set is obtained in a similar way, i.e., the pointer set is the intersection of three pointer sets, each is the result of checking one of the three predicates in the query. Again, if the intersected pointer set is empty, then there are no qualified tuples under this middle node. Otherwise, search process follows pointers in this pointer set to access qualified leaf nodes. All tuples pointed to by a qualified leaf node are loaded from the disk and values in A_t, A_2, and A_3 of the loaded tuples are checked directly to determine the exact set of qualified tuples for the query.

For clarity, we present an algorithm for conducting search in a three-level MG-tree as above. Assume no predicate involves the attribute (A_t) at which the MG-tree is established.

1. hash the value in each predicate (corresponds to an attribute) of the query into a query signature.

2. each query signature is used to scan 50 group signatures in the same attribute of the query signature in the top node to obtain a pointer set.

3. all pointer sets in the top node are intersected to produce a pointer set pt(top).

4. for each middle level node md in NODE(pt(top)):

    (a) each query signature is used to scan 50 group signatures in the same attribute of the query signature in md to obtain a pointer set.

    (b) all pointer sets in each md are intersected to produce a pointer pt(md).

    (c) for each leaf node ld in NODE(pt(md))

        i. retrieve all tuples pointed to by pointers in the leaf node into the memory.

        ii. use each predicate to check each of these tuples to find the true query result.

As a comparison, if three B+ trees are used for attributes A_t, A_2, and A_3, then each of the three predicates has to be evaluated separately and has a set of qualified tuples. The true query result is the intersection of the three tuple sets. In fact, the intersection can be done for the pointers and tuples are loaded only after pointer sets are intersected. Even with this improvement, in comparison to the MG-tree approach, this intersect operation happens at the very end of the query execution and the pruning is much less effective than doing the intersection at the higher levels of the tree as in the MG-tree. Some query optimizers may choose to use less than three B+ trees. For example, one of the attribute is accessed through a B+ tree and the selected tuple set is loaded into the memory to be linearly scanned using the other two predicates. If the selected set of tuple is large or the average tuple size is large, this practice will have much slower performance than using three B+ trees.

Some may concern that a non-leaf node of a MG-tree is much larger than a non-leaf node of a B+ tree. However, large non-leaf nodes only cause a negligible increase in the quantity of data that need to be loaded into the memory. For example, in the above example, the MG-tree of a 125k tuple dataset (if each tuple has 1k bytes, than the size of the data set is 125MB) has only 50 middle level nodes and one top level node. On the other hand, much less leaf and non-leaf nodes need to be accessed in a MG-tree than in B+ trees due to stronger pruning effect caused by intersection of pointer sets at non-leaf nodes of the MG-tree and this saving much exceeds the increase of the size of individual non-leaf nodes. This will be analyzed in detail in Section 4.

One may also concern that more CPU computations are needed to process a non-leaf MG-tree node than a non-leaf B+ tree node. For space limitation, we will not present a detailed CPU time comparison in this paper. However, typically the CPU time is only a fraction of the disk I/O time. Again, if we take the example above, there are only 51 non-leaf nodes in the MG-tree and the increase in CPU time (the extra thinking time at 51 non-leaf nodes) should be negligible as compared to the savings (due to pruning) in disk I/O time.

In a MG-tree, meta-data from each attribute is not mixed with meta-data from other attributes, this unique arrangement allows the intersection of pointer sets $P1$, $P2$, and $P3$ to be done early in the search process to enhance the pruning effect. The comparison of sizes and pruning effects between MG-tree and previous multi-dimensional signature methods will be presented in Section 5.

Maintaining an MG-tree: As any index structure, MG-tree need to be maintained when dataset is updated. Here we recommend some ways to save the amount of maintenance work for the MG-tree.

For deletion, if only a few tuples deleted in the dataset, then the MG-tree does not need to be updated because deleting a tuple only slightly decreases the search efficiency in the MG-tree. A threshold can be set to determine when the MG-tree needs to be updated. For example, if the fan out f=50, the threshold can be set as for at least one leaf node, 10 out of its 50 tuples are deleted. When this condition is satisfied, then this leaf node and all nodes above this leaf nodes which are the parents (recursively) of this leaf nodes need are updated. So, updates due to deletion can be delayed in the MG-tree to save some maintenance work.

For addition, similar delay tactic can be employed to save update work for the MG-tree. A dedicated "input partition" of tuples of the dataset can be set up to accept newly added tuples. The input partition is outside the MG-tree. A special "characteristic" tuple with length $t-1$ (assume R has t attributes) is created for the input partition. For each attribute where signatures are made for the MG-tree, one group signature is made out of values in tuples in the input partition for this attribute and is entered in the characteristic tuple. Similar to MG-tree, the meta-data in the characteristic tuple can be used to help search values in the input partition. When the size of the input partition reaches the average partition size (fan out of a leaf node) f, the input partition is disbanded and its tuples are inserted into tuple partitions of appropriate leaf nodes. The allocation of these tuples is according to values of A_t, the attribute at which the MG-tree is created, of each tuple in the disbanded input partition. So, updates due to addition can also be delayed in the MG-tree to save some maintenance work.

Only a value change in the dataset causes an immediate update of the MG-tree. If a value is located at attribute A and tuple S, which is under leaf node N, which in turn is under a second level non-leaf node NS, then, the group signature at (N,A) of NS is re-made. If there are higher level non-leaf nodes, the updated group signature will be used to further update signatures at higher levels. As our example shows, A MG-tree with two to three levels of nodes can index a very large dataset and so is practically useful. In this case, a value change in the dataset only causes one or two group signatures to be updated.

We would like to emphasize that the workload of maintaining the MG-tree is separate from the workload of query executions.

3 False Drop Ratio and the Size of an MG-Tree

In practice, the challenge of applying signature index is to avoid obtaining ones-only signatures at the top level of the MG-tree and the mostly ones-only signatures at the level below to insure that nodes at each level have certain filtering effect (a ones-only signature has no filtering effect at all). Longer signatures have more zeros and better filtering effect. We present a detailed false drop ratio analysis to show that an MG-tree with reasonable size can have sufficient filtering effect.

Size estimation: For simplicity of explanation, we assume that each non-leaf node has exactly f pointers and each leaf node has exactly m pointers. Table 1 lists all parameters used in the analyses. All performance analyses and numerical experiments are done for a tabular dataset R with n tuples and t attributes.

An MG-tree T is established at only one attribute A_t in R and all other attributes contribute signatures to T. Each leaf node in T has $m-1$ values in the data domain of A_t, and m pointers pointing to tuples of R. Each non-leaf node has $(t-1)f$ signatures, $f-1$ values in the data domain of A_t, and f pointers pointing to lower level nodes. In the following discussion, all signatures of a particular attribute at a particular level are referred to as a signature file. Thus, each non-leaf level has $t-1$ signature files belonging to $t-1$ attributes, respectively.

Taking into account that each i-th level (i=1 is the top level) signature is obtained from mf^{h-i-1} data items, the length of a signature in a i-th level node is approximately ([5]):

$$b_i = (1/log_e 2)^2 m f^{h-i-1} log_e(1/F_i), \qquad 1 \le i \le h-1. \tag{1}$$

So, the total size of all signatures of an attribute can be estimated as (in bytes):

$$\begin{aligned}\hat{O}_s &= \tfrac{1}{8}\sum_{i=1}^{h-1}\left(\tfrac{n}{mf^{h-i-1}}\right)b_i \\ &= \tfrac{1}{8}\sum_{i=1}^{h-1}\left(\tfrac{n}{mf^{h-i-1}}\right)(1/log_e 2)^2 m f^{h-i-1} log(1/F_i) \\ &= \tfrac{1}{8}n(1/log_e 2)^2 log_e(1/\prod_{i=1}^{h-1} F_i)\end{aligned} \tag{2}$$

From equation (2), it can be seen that approximately, the size of all signatures in T is only determined by the global false drop ratio $F = \prod_{i=1}^{h-1} F_i$.

Table 1: Parameters For Size and False drop Computation

h	the number of levels in the MG-tree T.
t	number of attributes in R.
f	fan-out of nodes in the MG-tree.
m	partition size.
$\hat{m}$	number of distinct values in a partition.
n	number of tuples in R, $mf^{h-1} = n$.
d	number of distinct values in an attribute, $d \leq n$.
b_i	length of i-th level signatures, in bits.
F_i	false drop ratio of i-th level signatures.
F	global false drop ratio, $F = \prod_{i=1}^{h-1} F_i$
s_v	size of a value, assume to be 6 bytes.
s_p	size of a pointer, assume to be 4 bytes.
O_s	space overhead of an MG-tree
O_B	space overhead of all B+ trees

If all data items in a partition have distinct values, the effective partition size $\hat{m}$ is equal to m. The existence of duplicate values in an attribute reduces the effective partition size $\hat{m}$. On the average, each unique value v has about n/d occurrences and they are randomly distributed over n/m partitions. By re-interpreting the meaning of parameters in Yao's equation [2], we can compute the number of partitions that contain at least one data item of v as:

$$Yao(\frac{n}{d}, \frac{n}{m}, n) = \frac{n}{m} - \frac{n}{m} * \prod_{i=1}^{\frac{n}{d}} \frac{n-m-i+1}{n-i+1}. \tag{3}$$

Thus, the average number of occurrences of v in a partition which contains v is:

$$s = (\frac{n}{d})/Yao(\frac{n}{d}, \frac{n}{m}, n). \tag{4}$$

So, the effective partition size $\hat{m}$ can be computed as:

$$\hat{m} \quad = m/s \quad = (\tfrac{m*d}{n})Yao(\tfrac{n}{d}, \tfrac{n}{m}, n) \tag{5}$$

Substituting $\hat{m}$ for m in equation (1), and taking into account that an MG-tree has one

B+ tree and $(t-1)$ signature files, the size of the MG-tree T is (in bytes):

$$\begin{aligned}
O_s &= \sum_{i=1}^{h-1}(\tfrac{n}{mf^{h-i-1}})(t-1)b_i/8 + \sum_{i=1}^{h-1}(\tfrac{n}{mf^{h-i}})(s_v*(f-1)+s_v*f) \\
&\quad +((n-1)*s_v+n*s_p) \\
&\approx \sum_{i=1}^{h-1}(\tfrac{n}{mf^{h-i-1}})(t-1)(1/log_e 2)^2\hat{m}f^{h-i-1}log(1/F_i)/8+n(s_v+s_p) \\
&\quad +(\tfrac{n}{m})(s_v+s_p)\sum_{i=1}^{h-1}(\tfrac{1}{f^{h-i-1}}) \qquad (6) \\
&= (\tfrac{n(t-1)}{8})(\tfrac{\hat{m}}{m})(1/log_e 2)^2 log_e(1/\prod_{i=1}^{h-1}F_i) \\
&\quad +n(s_v+s_p)(1+(\tfrac{1}{m})(1-\tfrac{1}{f^{h-1}})/(1-\tfrac{1}{f})) \\
&\approx (\tfrac{n(t-1)}{8})(\tfrac{\hat{m}}{m})(1/log_e 2)^2 log_e(1/\prod_{i=1}^{h-1}F_i)+n(s_v+s_p)
\end{aligned}$$

The above results is obtained under a uniform distribution assumption. If there is a data skew, then the result from equation (3) should be smaller and $\hat{m}$ will also be smaller. This will cause a smaller O_s by equation (6). Also, the second term $n(s_v+s_p)$ in equation (6) may be eliminated if the tuples in R are sorted according to values in A_t. Unlike the situation in a fully inverted file, this sorting does not cause performance penalty for accessing data in other attributes because the MG-tree only uses one B+ tree. If tuples are sorted according to values in A_t, the attribute at which the MG-tree is established, then it is called a *sorted MG-tree*.

In comparison, the size of all B+ trees in a fully inverted file is (in bytes):

$$\begin{aligned}
O_B &= nt(s_v+s_p)(1+(\tfrac{1}{m})(1-\tfrac{1}{f^{h-2}})/(1-\tfrac{1}{f})) \\
&\approx nt(s_v+s_p)
\end{aligned} \qquad (7)$$

The size reduction achieved by using a sorted MG-tree over t B+ trees is:

$$(O_B-O_s)/O_B \approx 1-\tfrac{1}{t}-(\tfrac{1}{8(s_v+s_p)(log_e 2)^2})(\tfrac{t-1}{t})(\tfrac{\hat{m}}{m})log_e(1/\prod_{i=1}^{h-1}F_i) \qquad (8)$$

It worth mentioning that the global false drop ratio $\prod_{i=1}^{h-1}F_i$ is only for one attribute. For a query that has q conjunctive predicates, the actual false drop ratio for searching the signatures in the MG-tree is $(\prod_{i=1}^{h-1}F_i)^q$, assuming signature files of different attributes have the identical false drop ratios. So, equation (8) gives a conservative estimate of the index size reduction.

Experiments and computation results: Each experiment is done on a dataset with n=125k data items. These 125k data items can be values of an attribute in 125k tuples. If each tuple has 40 attributes and each data entry in a tuple has an average of 30 bytes, then

125k data items represent a dataset of 153MB. These 125k data items are partitioned into 2,500 partitions each having m=50 data items. In experiments related to Fig.5, 6, and 7, each data item is represented by a 300-bit 0/1 string with four uniformly distributed 1-bits. For each experiment, d distinct values are generated. d varies from 30 to 1,000. We randomly select 20 out of d data items as query signatures. For the generation of the dataset, each partition is made by randomly selecting 50 (not necessarily distinct) data items out of d distinct values. These 50 signatures are bitwise-ORed to obtain a partition signature. Thus we obtain a file of 2,500 partition signatures, which are meta-data of a 153MB dataset. When generating the dataset, each time a data item, i.e., a signature is generated, it is checked against each of the 20 query signatures. The number of matches obtained is recorded as the number of true drops. No more than one true drop is recorded for each query signature and partition pair.

During a query execution, each query signature is checked against every partition signature and the number of matches are recorded as the number of total drops. The number of false drops is obtained by subtracting the number of true drops (obtained in the signature file generating phase) from the number of total drops (obtained in the query execution phase). The experiment is run on a HP workstation using library routines drand48() and srand48(). Each plotted data point is the average of the results from 40 experiments and in each experiment the results is the average of 20 queries. Different experiments use different seeds for the random number generators.

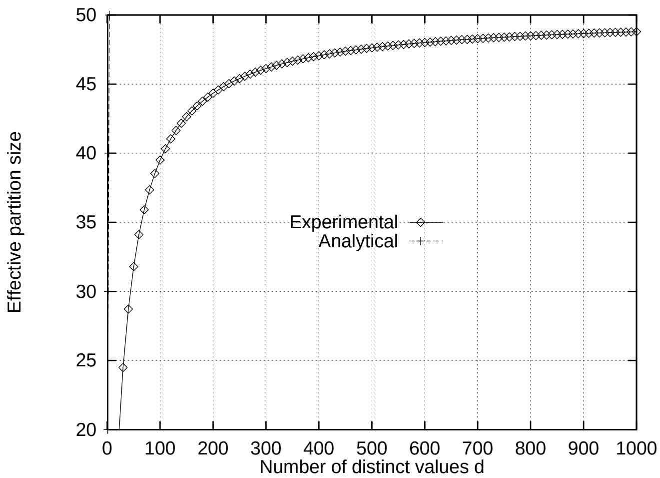

Figure 5: $\hat{m}$ vs. d.

Fig.5 shows the change in the effective partition size $\hat{m}$ with respect to d, the total number of distinct values. The analytical results are obtained by using equation (5) and are nearly indistinguishable to the experimental results obtained by direct counting in experiments.

Fig.6 shows the change in the false drop ratio F with respect to d. The analytical results

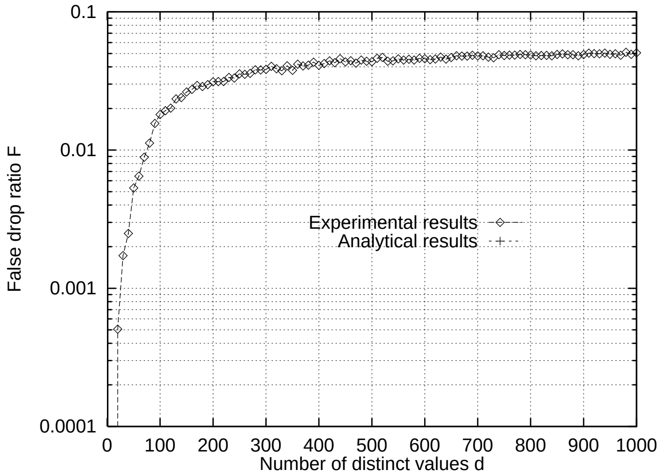

Figure 6: F vs. d.

are very close to the experimental results. When d is large, e.g, $d > 500$, F stablizes at about 0.05. This shows that when d is much larger than the partition size, the effect of duplicate values diminishes.

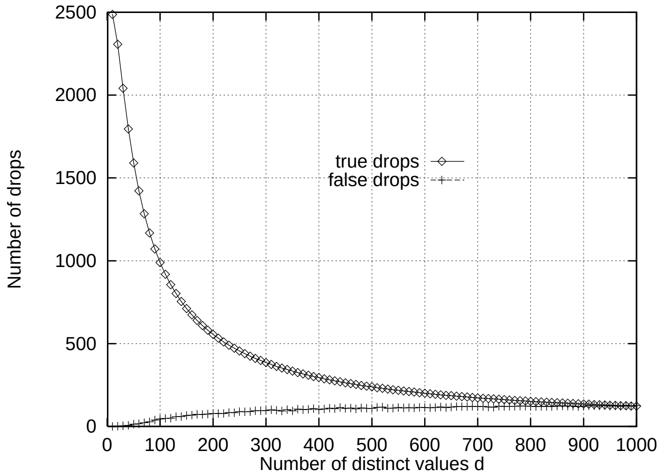

Figure 7: Experimental results of the number of true and false drops.

Not only the false drop ratio is low when the number of distinct values d is low, from Fig.7, we can also see that as d decreases, the number of true drops increases and the number of false drops decreases. When d is small, for example, $d = 100$, the ratio of false drops to the true drops is very low and the effect of false drops can almost be ignored.

The relation between signature length (and so the size of the signature file) and false drop ratio has been been studied before, though not in the context of a small d. For completeness, we show in Fig.8 both the analytical results using equation (1) and the experimental results, with $d = 400$. It can been seen that the difference between the analytical and the

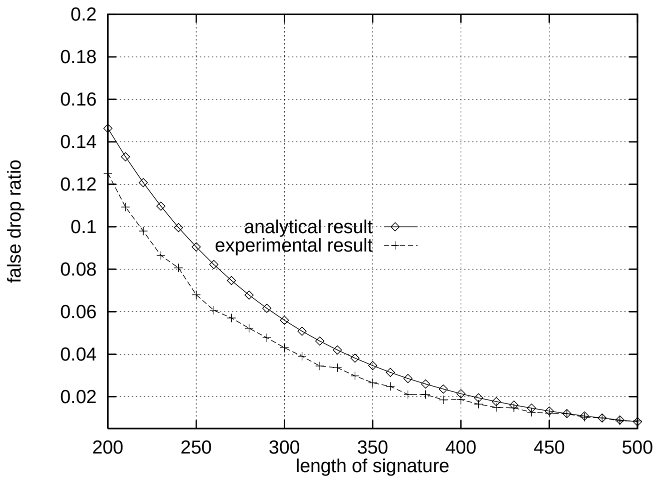

Figure 8: F vs. the length of signatures, d=400.

experimental results is small.

For a signature method to be effective, the false drop ratio must be low enough with a not too big signature length. Note that the false drop ratios plotted in Fig.8 is just for a single attribute at a single level. If an MG-tree has three levels (so its signature file has two levels) and a query has three conjunctive predicates, the resulting overall false drop ratio is (refer to Fig.8) at most $(0.05)^{2*3} = 1.6 * 10^{-8}$, which is a sufficiently low figure in most applications.

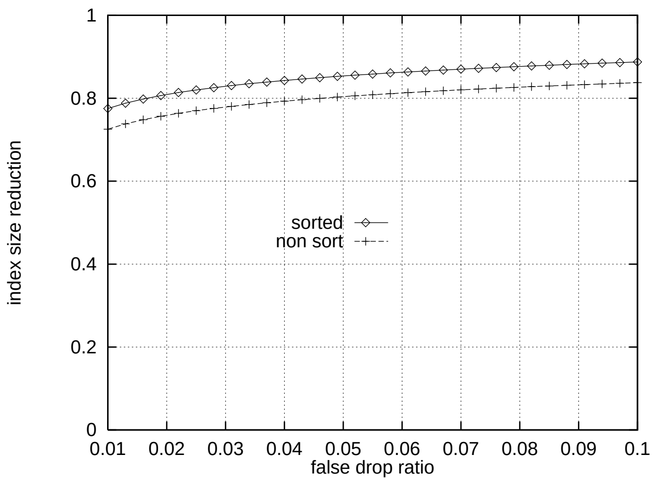

Figure 9: Index size reduction ratio from 20 B+ trees to an MG-tree.

Finally, we use equation (8) to plot the size reduction ratios in Fig.9. In Fig.9, a sorted MG-tree has a higher (better) reduction ratio than a non-sorted MG-tree because the sorted MG-tree is smaller than a corresponding non-sorted one.

Table 2: Parameters For Access Speed Computation

k_i	number of 1-bits of signatures hashed out for level i.
q	number of conjunctive predicates in a conjunctive query.
t_j	number of qualified data items in j-th attribute, t_j=10.
Q_v	number of data value to be checked for a query.
G_i	number of signatures searched at level i.
$G_{T,i}$	number of i-th level signatures matched due to true drops.
$G_{F,i}$	number of i-th level signatures matched due to false drops.
$L_{S,i}$	size of i-th level MG-tree nodes loaded, in bytes.
L_{extra}	size of extra tuples due to false drops, in bytes.
$L_{B,i}$	size of all i-th level B+ tree nodes loaded, in bytes.
s_{ID}	size of a tuple ID, which is assumed to be 6 bytes.
s_{tuple}	size of a tuple, which is assumed to be 1024 bytes.
$L_{B,ID}$	size of all tuples IDs loaded in B+ tree method, in bytes.
$B_{M,i}$	number of i-level B+ tree nodes matched.
B_i	number of i-th level B+ tree nodes loaded, in bytes.

4 Search Effort Comparison

We measure the search effort by the size of total MG-tree or B+ tree nodes that need to be loaded into the memory during the search and the search effort is compared between an MG-tree and multiple B+ trees for the same query containing q conjunctive predicates. The attributes involved in the query are A_i, $1 \le i \le q$. For simplicity of the analysis, we assume each non-leaf node of the MG-tree or a B+ tree has exactly the same fan out f and each leaf node has pointers pointing to exactly m tuples.

Table 2 lists all parameters used in the search effort analyses. Note that k_i is not the number of 1-bits in each of the i-th level signatures. Each of the i-th level signature is obtained by bitwise-ORing signatures that have k_i 1-bits. The number of signatures searched at level i (level 1 is the top level) for each predicate in a query having q conjunctive predicates can be computed as:

$$
\begin{aligned}
G_i &= \begin{cases} f & i = 1 \\ (G_{T,i-1} + G_{F,i-1})f & 1 < i \le h \end{cases} \\
G_{T,i} &= f^i \prod_{j=1}^{q} (1 - (1 - \tfrac{1}{f^i})^{t_j}) \qquad 1 \le i < h \\
G_{F,i} &= \begin{cases} (f - G_{T,1})F_1^q & i = 1 \\ (G_i - G_{T,i})F_i^q & 1 < i < h \end{cases}
\end{aligned} \tag{9}
$$

Value distribution in each of the q attributes is assumed to be random and independent. So, the probability that a pointer is qualified (there are q signatures need to be checked for each pointer in a node) is the product of q factors for both true and false drops in equation (9). The size of the portion of the i-th level MG-tree nodes which need to be loaded into the memory is (in bytes):

$$L_{S,i} = \begin{cases} b_1 qf/8 + s_v(f-1) + s_p f & i = 1 \\ b_i q G_i/8 + (s_v(f-1) + s_p f)G_i & 1 < i < h \\ (s_v(m-1) + s_p m)G_h & i = h \end{cases} \tag{10}$$

Note that bit-sliced format cannot be used here because each signature resides in a separate node of the MG-tree. The extra tuples need to be loaded due to false drops should also be taken into account for the search effort in an MG-tree. This can be computed as:

$$L_{extra} = G_{F,h-1} * m * s_{tuple} \tag{11}$$

In comparison, if q B+ trees are used to answer the conjunctive query, the number of the i-th level nodes for the j-th B+ tree ($1 \le j \le q$) that need to be loaded into the memory is:

$$\begin{aligned} B_i &= \begin{cases} 1 & i = 1 \\ B_{M,i-1,j} f & 1 < i \le h \end{cases} \\ B_{M,i,j} &= f_i(1 - (1 - \tfrac{1}{f^i})^{t_j}) \qquad 1 \le i < h \end{aligned} \tag{12}$$

So, the size of the portion of the i-th level nodes of all q B+ trees which needs to be loaded into the memory can be computed as (in bytes):

$$L_{B,i} = \begin{cases} qB_i(s_v(f-1) + s_p f) & 1 \le i < h \\ qB_i(s_v(m-1) + s_p m) & i = h \end{cases} \tag{13}$$

The common part of the q sets of tuples obtained from accessing q B+ trees is the true answer to the conjunctive query. We assume that the performance for using B+ trees to answer conjunctive queries is enhanced by a mechanism which makes the intersection of all q sets of tuples IDs (their size is much smaller than the size of all q sets of tuples) before actual tuples of R are accessed. This mechanism can greatly reduce the number of disk accesses needed. The size of q sets of tuple IDs obtained from accessing q B+ trees is:

$$L_{B,ID} = \sum_{j=1}^{q} t_j s_{ID} \tag{14}$$

Fig.10 shows the comparison of the search effort when using an MG-tree with the search effort when using multiple B+ trees to answer conjunctive queries. The false drop ratio per level per attribute is varied from 0.01 to 0.1. Each tree has three (h=3) levels. The queries have q=1, 2, or 3 conjunctive predicates. When q=1, using a single B+ tree clearly

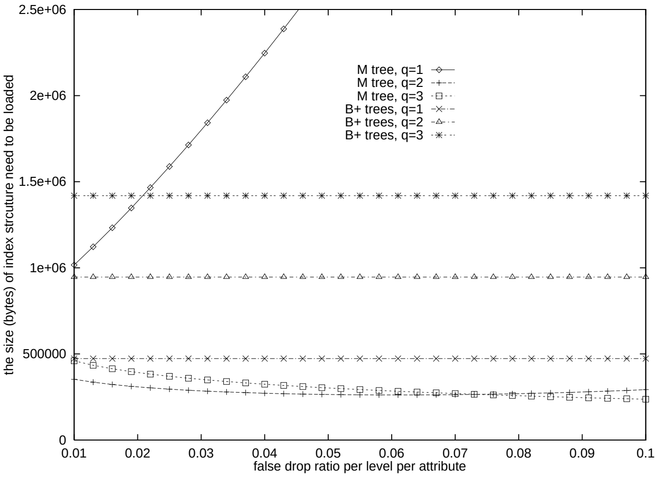

Figure 10: Index size loaded in an MG-tree and in multiple B+ trees

outperforms using an MG-tree at an attribute other than the attribute of the MG-tree (so, signature files instead of the B+ tree in the MG-tree is used). However, when q=2 or 3, using an MG-tree, even at attributes other than the attribute of the MG-tree and even with the enhancement to the B+ trees, using an MG-tree outperforms using two or three B+ trees.

While the search effort in an MG-tree is much larger than the search effort in a B+ tree in the case of q=1, i.e., non-conjunctive query, we should keep in mind that a tabular dataset indexed by an MG-tree is much smaller than a fully inverted file with every attribute indexed by a B+ tree. Refer to Fig.9, the corresponding index size reduction ratio for a non-sort MG-tree over 20 B+ trees on a 20 attribute dataset is from 72% to 84%.

5 Related Work

One may wonder whether we can combine several tree-like indexes to achieve similar effect as an MG-tree (After all, tree-like indexes have higher data access speed than signature files). This is not feasible since each tree-like index has its own partitioning of tuples and partitionings of different tree-like indexes are likely to conflict with each other. Only signature files have the flexibility to work with arbitrary partitions because a bitwise-ORed signature can be obtained from an arbitrary partition.

Multi-level signature files have been studied previously [4, 5, 8, 16] to achieve faster access than linear scan of signature files. The difference of the MG-tree is that it combines together many multi-level signature files from many attributes with a B+ tree. The B+ tree in the MG-tree serves four purposes: it provides a partitioning (note that no physical clustering of tuples are needed) of the tuples in the tabular dataset; it provides an organized storage for these signature files and physically binds them together; it serves as a shared (so is an efficient) pointer structure for these signature files, and itself is also used as a fast access index. This novel arrangement enables the MG-tree to achieve fast processing of conjunctive queries for tabular datasets.

The multi-level multi-dimensional signature indexing described by Chang and Schek [11] was the closest method to the MG-tree. They use tuple signatures. A tuple signature is a group signature obtained by bitwise-ORing signatures generated from every field of a tuple. In constructing an MG-tree, we bitwise-OR signatures along each attribute.

Our MG-tree method has the following advantages over the tuple signature approach:

1. Bitwise-ORing along each attribute simplifies the implementation. In a real-life database, different attributes contains different types of data, numbers, objects, text, sound segments, images, etc. To make a tuple signature needs to bitwise-OR signatures generated from different data types. Different data types use different hash functions and may have different density of "1" bits in the generated signatures, yet the tuple signature must have proper density and distribution of "1"s and "0"s to ensure good filtering effect. This is much more difficult to do in practice than bitwise-ORing signatures in the same attribute and to ensure proper density and distribution of "1"s and "0"s in the resulting group signature.

   Also, a real-life database has many different tabular datasets (relations). Each dataset may have different dimensions. For example, one tabular dataset may have only 7 attributes and another may have 100 attributes. For each tabular dataset, tuple signatures should have proper density and distribution of "1"s and "0"s to ensure a good filtering effect. This means in practice, hash functions, even for the same data type, have to be adjusted for each individual dataset. This represents a lot of work for practical system implementors. This problem does not exist for MG-tree. This advantage makes the implementation of an MG-tree much more consistent and robust across different database instances than previously proposed multi-dimensional signature indexing techniques.

2. Bitwise-ORing along each attribute reduces the search work. Only signatures in those attributes that involved in the query predicates are searched. This presents a drastic reduction in search work. For example, if a tabular dataset has 40 attributes and the query only has three conjunctive predicates, then only $3/40 = 7.5\%$ of group signatures are searched in an MG-tree compared to the tuple signature approach in [11].

   This search effort reduction is especially important for small selectivity (small search result) queries because for these queries the amount of search work in the signature file, whose size may be 10% of the size of the entire dataset, becomes significant and even dominant in the query execution effort.

We would like to emphasize that while it is important that the MG-tree outperforms the tuple signature approach [11] for small selectivity queries, the ease and stability of implementation alone can make a difference between a research interest and a usable technology in real-life DBMS implementations.

6 Concluding Remarks

In this paper, we introduced the concept of the MG-tree which is made by utilizing the flexibility of signature files and the efficiency of the pointer structure of a tree-like index. The MG-tree combines the strength of both signature files (small size) and tree-like index (faster access). Because of the novel arrangement in each non-leaf node, the MG-tree can even achieve data access faster than multiple tree-like indexes for queries with two or more conjunctive predicates. Analyses and experiments have been conducted to verify the performance of the MG-tree.

Compared to previous proposed multi-dimensional signature indexing techniques [10, 11], the implementation of an MG-tree is much easier and robust, and the search speed of an MG-tree is much faster, especially in executing small selectivity queries. We believe these two advantages makes the MG-tree an practically useful technology to the actual DBMS implementation.

The performance advantage and the small size of the MG-tree make it a good alternative to a fully inverted file approach for fast data access in tabular datasets such as relations in relational databases, especially if conjunctive queries ($q > 1$) are frequently executed.

The MG-tree is suited for use in relational databases. With the vast number of installations of relational databases today and their critical role in businesses of corporations, the design of practical indexes such as the MG-tree is still a field that well worth our attention.

References

[1] R. Bayer, E.M. McCreight,"Organization and Maintenance of Large Ordered Indexes," *Acta Informatica 1972, pp. 173-189.*

[2] S.B. Yao. "Approximating Block Accesses in Database Organizations," *Communications of ACM Vol. 20, No. 4, pp. 260-261, April 1977.*

[3] C.S. Roberts, "Partial-match retrieval via method of superimposed codes," *Proc. IEEE, Vol. 67, No. 12, Dec. 1979, pp. 1624-1642.*

[4] J.L. Pfalts, W.J. Berman, E.M. Cagley, "Partial-Match Retrieval Using Indexed Descriptor Files," *CACM 1980, pp. 522-528.*

[5] R. Sacks-Davis, "A Two Level Superimposed Coding Scheme For Partial Match Retrieval," *Information Systems, Vol. 8, No. 4, 1983, pp. 273-280.*

[6] S. Christodoulakis, C. Faloutsos, "Design considerations for a message file server," *IEEE Transactions on Software Engineering, Vol. SE-10, No. 2, March, 1984, pp. 201-210.*

[7] C. Faloutsos, S. Christodoulakis, "Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation," *ACM Transactions on Office Information Systems, Vol. 2, No. 4, October 1984, pp. 267-288.*

[8] U. Deppisch, "S-Tree: A Dynamic Balanced Signature Index for Office Retrieval," *Proc of the ACM Conference on Research and Development in Information Retrieval, Pisa, Italy, Sept. 1986, pp. 77-87.*

[9] C. Faloutsos, S. Christodoulakis, "Description and Performance analysis of signature file methods for office filing," *ACM Transactions on Office Information Systems, Vol. 5, No. 3, July 1987, pp. 237-257.*

[10] R. Sacks-Davis, A. Kent, K. Ramamohanarao, "Multikey Access Methods Based on Superimposed Coding Techniques," *ACM Transactions on Database Systems, Vol. 12, No. 4, Dec. 1987, pp. 655-696.*

[11] W.W. Chang, H.J. Schek, "A Signature Access Method for the Starburst Database System," *Proc. of the 15th Int'l Conf. on Very Large Data Bases, 1989, pp. 145-153.*

[12] D.L. Lee, C.-W. Leng, "Partitioned Signature Files: Design Issues and Performance Evaluation," *ACM Transactions on Information Systems, Vol. 7, No. 2, April 1989, pp. 158-180.*

[13] C. Faloutsos, "Signature-based text retrieval methods: A survey," *Data Engineering Bulletin, Vol. 13, No. 1, March 1990, pp. 25-32.*

[14] D.L. Lee, C.-W. Leng, "A partitioned signature file structure for multiattribute and text retrieval," *Proceedings of the 6th Int'l Conf. on Data Engineering, Los Angeles, Feb. 1990, pp. 389-397.*

[15] Z. Lin, C. Faloutsos, "Frame-Sliced Signature Files," *Transactions on Knowledge and Data Engineering, Vol. 4, No. 3, June 1992, pp. 281-289.*

[16] D.L. Lee, Y.M. Kim, G. Patel, "Efficient Signature File Methods for Text Retrieval," *IEEE Transactions on Data and Knowledge Engineering, Vol. 7, No. 3, June 1995, pp. 423-435.*