M-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

July 31, 1998

Abstract

A novel index structure, M-tree, which combines the strength of both tree-based indexes and signature files is introduced in this paper. An M-tree is used for indexing data in multiple attributes of a tabular dataset, but it is created at one attribute and its signatures are collected from several other attributes. One M-tree can replace several tree-based indexes and is much smaller than the tree-based indexes it replaces. Because signatures of several attributes co-exist in each node of an M-tree, for answering conjunctive queries, accessing data through an M-tree can even be faster than accessing data through several tree-based indexes. Analyses and numerical experiments are conducted to evaluate the storage space and access speed of M-trees.

Some aspects of the practical implementation of signature method in general database environments are also considered in this paper. These include applying signature method to indexing numerical data and duplicate data.

Key words: Signature, access index, ordered data, conjunctive query.

1 Introduction

Tree-based access indexes such as B-trees [1] or B+ trees have attracted much interests in the past. They provide very fast access but have very large storage space overhead. These indexes have been used extensively in database implementations, notably in relational databases.

A relation in a relational database is a tabular dataset. In general, 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. A large tabular dataset may have millions or more tuples. A tree-based index created at an attribute of a tabular dataset can greatly reduce the search time for accessing data in that attribute. A tabular dataset which is equipped with one tree based-index for each of its attributes is called a fully inverted file. A fully inverted file has a very high access speed. But, it is well known that the storage space overhead of all the tree-based indexes in a fully inverted file is very large, often larger than the primary dataset itself.

Alternatively, signature file as an access method (see Section 2) has been studied extensively in the past, mostly in the context of text retrieval [3, 7, 9, 12, 11, 13]. Using a signature file to access data is generally slower than using a tree-based index because usually a large portion of the signature file has to be scanned during a data access. Searching through a signature file may also produce false drops and thus further slows down the access speed. The advantage of a signature file is that it is much simpler and smaller than a tree-based index. Often a signature file is only 10-20% the size of the primary dataset [6].

We combine the strengths of both a tree-based index and a set of signature files and propose a *multi-attribute signature enhanced tree*, M-tree, as an access index. A M-tree can be built from a B+ tree and can replace several B+ trees. The size of an M-tree is much smaller than the size of all B+ trees it replaces. An M-tree is especially good for answering conjunctive queries. A conjunctive query has predicates involving several attributes, such as the one below:

$$
\begin{array}{llll}
select & R.name, & R.telephone & \\
from & R & & \\
where & R.specialization & = \text{“}software\text{”} & and \\
 & R.title & = \text{“}engineer\text{”} & and \\
 & R.status & = \text{“}available\text{”}; &
\end{array}
$$

Previously, as far as we know, no research has been done on applying signature method to indexing numerical data. This may in part due to the fact that signatures are generated by hashing. Hashing does not preserve order among data items. Numerical data are often involved in range queries which require order to be used in answering queries. We solve this problem by introducing interval signatures.

This paper also address another issue that has not been addressed by previous research, that is, to apply the signature method for indexing a large dataset having a small number of distinct values, i.e., a small data range. For example, the attribute “STATE” in the US has

only 50 distinct values (states). The attribute "store location" in a sales database is likely to have a small data range. In practice, it is common that a relation in a database has many attributes each having a small data range. The limited number of distinct values may be distributed over millions of tuples in a tabular dataset.

This paper is organized as follows. Section 2 briefly describes the signature method as a background. Section 3 introduces the concept of the M-tree. Section 4 analyzes the relationship between the size of an M-tree and false drop ratios under realistic situation which is the existence of duplicate data, and both analytical and experimental results are presented. Section 5 compares the access speed between an M-tree and multiple B+ trees for answering conjunctive queries. Section 6 extends the signature method, and so the M-tree, to indexing numerical data. Section 7 discusses the difference between the M-tree and some of the related works. Finally, Section 8 summarizes the paper.

2 Background: Signature Method

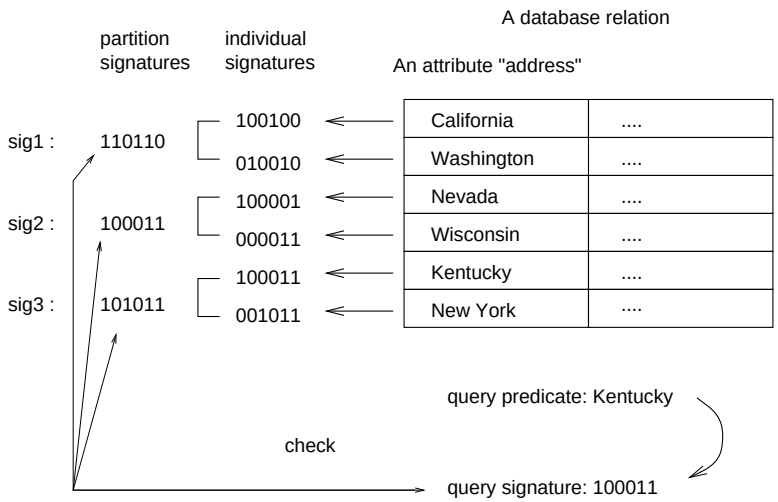

Figure 1: Using signatures to process a query.

In its simplest form, a *signature* is a 0/1 bit string obtained by hashing data items. Fig.1 shows a relation with an attribute "address". Each value in "address" is hashed into a signature. These signatures are partitioned and all signatures in a partition are superimposed (bitwise-ORed) to obtain a partition signature for the partition. *sig1, sig2*, and *sig3* are partition signatures. Consider the query below:

$$\begin{array}{ll} select & R.name \\ from & R \\ where & R.address = \text{“}Kentucky\text{”} \end{array}$$

The value "*Kentucky*" in the query predicate is hashed into a query signature *qsig*. *qsig* is checked against each of the partition signatures. A *match* of *qsig* with a partition signature is defined as every 1-bit in *qsig* is matched by a 1-bit in the same position of the partition signature. A match indicates "*Kentucky*" might exist in the partition of the matching partition signature.

The partition of $sig1$ does not contain value "*Kentucky*" because *qsig* does not match with $sig1$. Although *qsig* matches with $sig2$, a further checking shows that this is a false drop. Only $sig3$ has a true drop with *qsig* because $sig3$ represents a group which contains "*Kentucky*".

A large number of data items can be excluded from search because their partition signature do not match query signature. Using a signature file can significantly speed up search process because the signature file is usually much smaller than the primary dataset itself. The search speed can be further improved by using multi-level signature files [4, 5, 8, 15] in that not even the entire signature file has to be searched during a query execution.

3 M-Tree

Though an M-tree can be built by augmenting signatures to any tree based index, we focus our discussion on B+ tree for its importance in real-life systems. Suppose a B+ tree is created at attribute A_t. The leaf nodes of the B+ tree contain values in the data domain of A_t and pointers 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 M-tree can be built from such a B+ tree.

Structure of an M-tree: Fig.2 shows a typical non-leaf node of an M-tree. The M-tree is created at attribute t and signatures are collected from attributes A_1 through A_{t-1}. Each attribute A_i, $1 \leq i \leq t-1$, has exact one signature for each pointer in the node. 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. When processing a query having predicates involving attribute A_{q_i}, $q_i < t$, only signatures belong to attribute A_{q_i} are accessed. If $q_i = t$, then values v_1, v_2, ..., v_{f-1} are also checked. The checking of signatures and/or values v_1 through v_{f-1} determines which pointer need to be further traversed for the query processing. A leaf node has only values and pointers, but no signatures. Pointers of a leaf node point directly to tuples of a tabular dataset R.

We use an example to further illustrate the detailed construction of an M-tree from a B+ tree. Assume R has 125k tuples and 20 attributes. A B+ tree T with three levels is

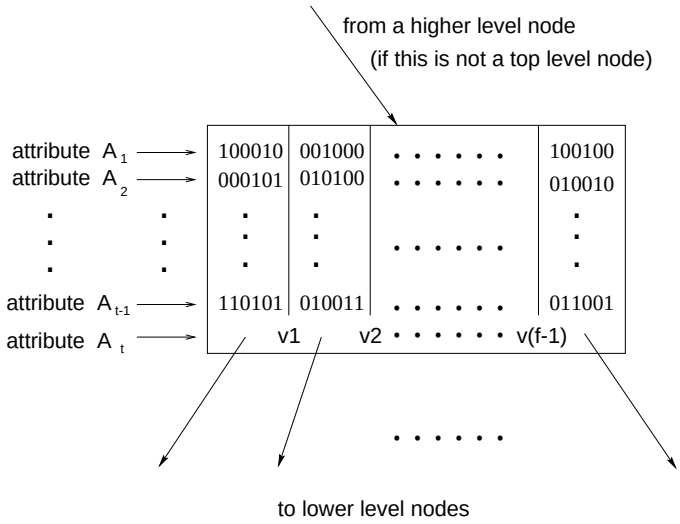

Figure 2: A typical non-leaf non-top node of an M-tree with t attributes.

constructed at A_t, t=20. 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.

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. So, 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 T is (2,500 + 50 + 1) $*$ (49 $*$ s_v + 50 s_p) $\approx 13 * 10^4(s_v + s_p)$ bytes.

Fast access is also desired for attribute A_1 through A_{19}. However, a fully inverted file may have too big an overhead for its access index. So, we use signatures to enhance the B+ tree so that it can provide fast access to attributes A_1 through A_{19}.

Let us focus our discussion on one attribute A_2. If it has a numerical data domain, then interval signatures are used as will be discussed in Section 6. Otherwise, traditional signatures are used. The 2,500 leaf nodes divide the tuples of R into 2,500 partitions as shown in Fig.3. The partitions of tuples induce partitions of data items in each of the attributes A_1 through A_{19}. For attribute A_2, every value in a partition can contribute (by a hash function ϕ_1) a signature towards the construction of the partition signature for attribute A_2. The partition signature of a partition is obtained by superimposing all signatures in that partition from attribute A_2.

The constructed partition signatures as described above are entered into middle level nodes in T. Each middle level node has 50 partition signatures because the fan out of each node is 50. Each partition signature corresponds to one pointer pointing to a bottom level

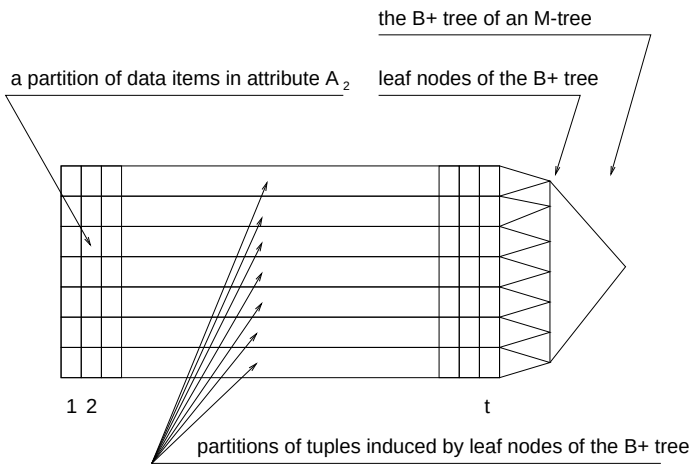

Figure 3: The partitioning of tuples by leaf nodes.

node. In each of the middle level node, there are total of (because there are 19 attributes, and each attribute has 50 signatures) $50 * 19 = 950$ signatures. Note that lead-nodes contain no signatures.

Several methods can be used to construct signatures for attribute A_2 at the top (the third) level. One way is to superimpose all signatures from middle level nodes to obtain signatures at the top level [4]. Each top level signature is superimposed from 50 middle level signatures of one 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 superimposing these signatures to obtain signatures at the top level [5, 15]. Each top level signature is superimposed from 2,500 signatures hashed from data items of one attribute.

Using an M-tree to answer conjunctive queries: When accessing data in attribute A_t, the M-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. The real performance advantage of using an M-tree is in answering conjunctive queries (see an example in Section 1) which search for tuples in R that simultaneously satisfy several predicates on several attributes. Conjunctive queries frequently appear in OLAP and have been receiving increasing attention in the research community.

Suppose a conjunctive query has three predicates: $R.A_t = a$, $R.A_2 = b$, and $R.A_3 = c$. Two query signatures S1 and S2 are made out of b and c, respectively.

By comparing value a with 49 values of A_t in the top level node, a set P1 of qualified pointers is selected. Each attribute other than A_t has 50 signatures in the top level node and each signature corresponds to a pointer pointing to a middle level node. Signature S1 is checked with the signatures from A_2 and the pointers of matched signatures are selected into

P2. Similar process can be done for S2 to obtain a qualified pointer set P3. If $P1 \cap P2 \cap P3 = \emptyset$, then search no more: the query will return an empty set. Otherwise, the search process 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 and a new set $P1 \cap P2 \cap P3$ is obtained for this node. If $P1 \cap P2 \cap P3 = \emptyset$, then there are no qualified tuples under this node. Otherwise, search process follows pointers in $P1 \cap P2 \cap P3$ of this node to obtain 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.

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 tuples sets. In comparison to the M-tree approach, this "AND" operation happens at the very end of the query execution and may result in a lot of tuples in B+ trees loaded unnecessarily.

When using an M-tree, the unique arrangement of signatures in the M-tree allows the intersection of pointer sets $P1$, $P2$, and $P3$ to be done early in the search process. So, this signature storage format has the potential to outperform the multiple B+ trees approach. Precise assessment of using an M-tree as compared to using multiple B+ trees in answering conjunctive queries is given in Section 5.

4 False Drop Ratio and the Size of an M-Tree

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 M-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]):

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

Table 1: Parameters For Size and False drop Computation

h	the number of levels in the M-tree T.
t	number of attributes in R.
f	fan-out of nodes in the M-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 M-tree
O_B	space overhead of all B+ trees

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

$$\begin{aligned} \hat{O_s} &= \frac{1}{8}\sum_{i=1}^{h-1}(\frac{n}{mf^{h-i-1}})b_i \\ &= \frac{1}{8}\sum_{i=1}^{h-1}(\frac{n}{mf^{h-i-1}})(1/log_e 2)^2 mf^{h-i-1}log(1/F_i) \\ &= \frac{1}{8}n(1/log_e 2)^2 log_e(1/\prod_{i=1}^{h-1} F_i) \end{aligned} \quad (2)$$

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 $F = \prod_{i=1}^{h-1} F_i$.

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:

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

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

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

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

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

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 M-tree only uses one B+ tree. If tuples are sorted according to values in A_t, the attribute at which the M-tree is established, then it is called a *sorted M-tree*.

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

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

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

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

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 M-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. The dataset is partitioned into 2,500 partitions each having m=50 data items. In experiments related to Fig.4, 5, and 6, 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 superimposed to obtain a partition signature. Thus we obtain a file of 2,500 partition signatures. 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.

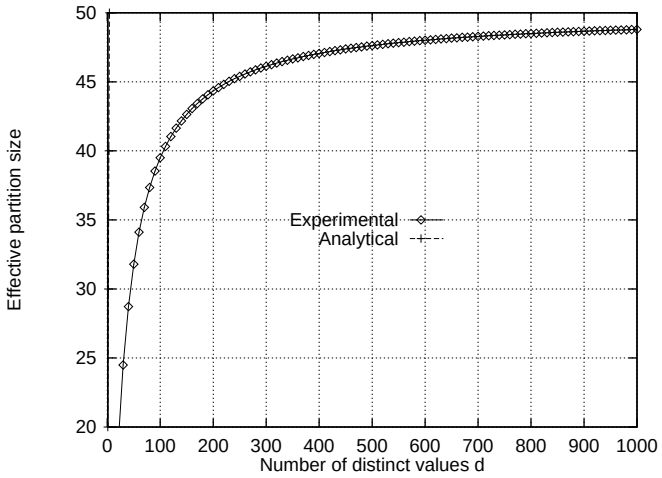

Figure 4: $\hat{m}$ vs. d.

Fig.4 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.5 shows the change in the false drop ratio F with respect to d. The analytical results

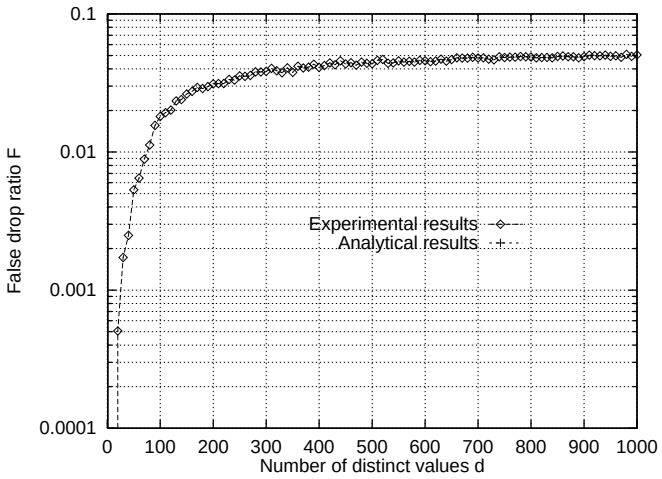

Figure 5: F vs. d.

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.

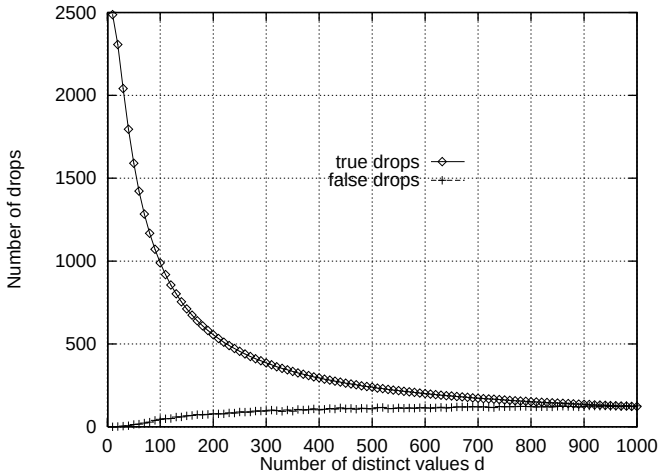

Figure 6: Experimental results of the number of true and false drops.

Not only the false drop ratio is low when the number of distinct values d is low, from Fig.6, 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.7 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

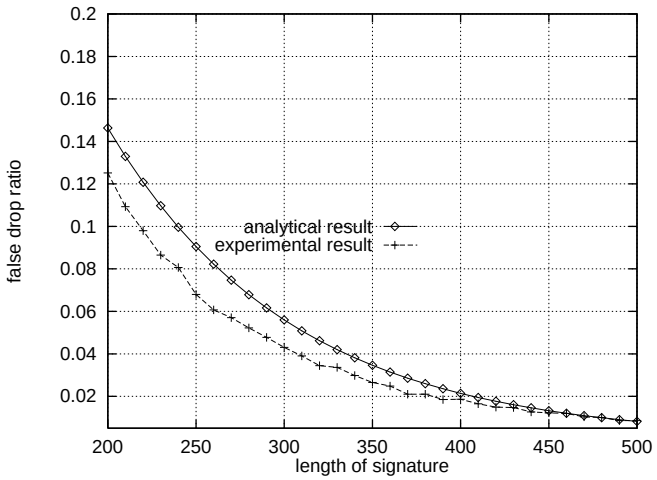

Figure 7: F vs. the length of signatures, d=400.

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.7 is just for a single attribute at a single level. If an M-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.7) at most $(0.05)^{2*3} = 1.6 * 10^{-8}$, which is a sufficiently low figure in most applications.

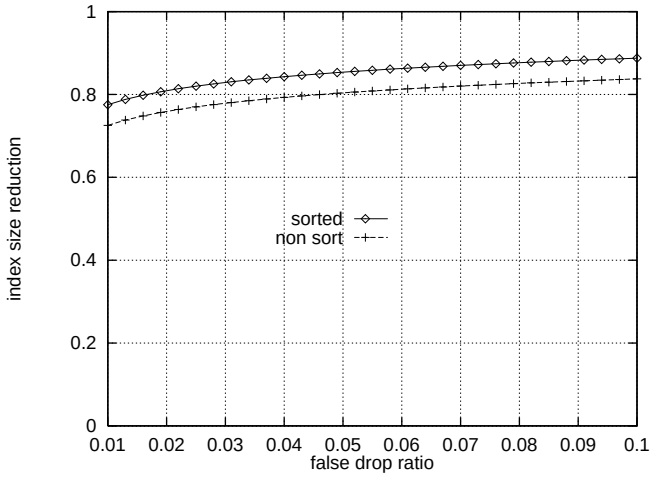

Figure 8: Index size reduction ratio from 20 B+ trees to an M-tree.

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

5 Search Effort Comparison

We measure the search effort by the size of total M-tree or B+ tree nodes that need to be loaded into the memory during the search and the search effort is compared between an M-tree and multiple B+ trees for the same query containing q conjunctive predicates. The attributes involved in the query are A_i, $1 \leq i \leq q$. For simplicity of the analysis, we assume eachnon-leaf node of the M-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 superimposing 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:

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

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 M-tree nodes which need to be loaded into the memory is (in bytes):

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

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

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

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 ($1 \leq j \leq q$) that need to be loaded into the memory is:

$$\begin{aligned} B_i &= \begin{cases} 1 & i = 1 \\ B_{M,i-1,j} f & 1 < i \leq h \end{cases} \\ B_{M,i,j} &= f_i(1 - (1 - \frac{1}{f^i})^{t_j}) \quad 1 \leq i < h \end{aligned} \quad (12)$$

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):

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

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:

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

Fig.9 shows the comparison of the search effort when using an M-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

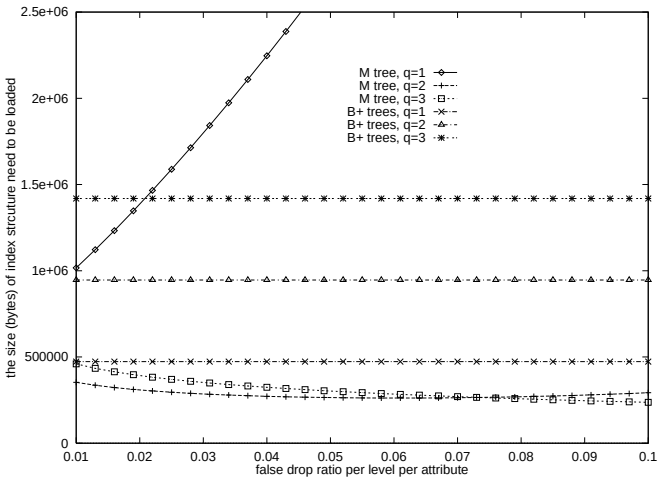

Figure 9: Index size loaded in an M-tree and in multiple B+ trees

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

While the search effort in an M-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 M-tree is much smaller than a fully inverted file with every attribute indexed by a B+ tree. Refer to Fig.8, the corresponding index size reduction ratio for a non-sort M-tree over 20 B+ trees on a 20 attribute dataset is from 72% to 84%.

6 Interval Signature

Previously, signature method has been studied only as an access index for text data because signatures are generated by hashing. Hashing does not preserve order among data. Numerical data are often involved in range queries and order information is important for range queries. For example, a query may look for all values in an attribute of a tabular dataset within a given interval $[v_q - 1, v_q + 1]$. This small interval contains infinite number of values and it is impossible to have one signature for each value in this interval. We propose to use interval signatures to solve this problem. Using interval signatures, M-trees can be used for accessing text data as well as numerical data.

We illustrate the concept of interval signature with an example. Assume a relation R has 100k tuples. The data domain of an attribute A_{i0} in R is a real valued interval [0,100]. The interval [0,100] is divided into 100 smaller intervals: [0,1], [1,2], ..., [99,100] and a hash function ϕ_{i0} is used for obtaining interval signature S[i,i+1] from interval [i,i+1], $0 \leq$ i $<$

100. For a value $v \in$ [i,i+1], the interval signature S[i,i+1] is called the *interval signature of* v. Suppose R has 100k tuples and 100k values in A_t are partitioned into 1,000 partitions of approximately equal size. A partition signature of a partition is formed by superimposing all (about 100 of them) interval signatures of the data items in the partition.

Suppose a query asks for all data items in A_{i0} that fall within interval $[v_q - 1, v_q + 1]$. If $v_q \in [i_q,i_q+1]$, then interval signatures of three intervals are used as query signatures: S[i_q-1,i_q], S[i_q,i_q+1], and S[i_q+1,i_q+2].

To answer this query, the signature file of 1,000 partition signatures is scanned. Each of these three query signatures is compared to every partition signature. A partition whose partition signature is not matched with any query signature contains no data items in $[i_q - 1, i_q + 2]$ and therefore is excluded from further search. The effort to search 1,000 partition signatures and then search partitions with matched partition signatures is much smaller than the effort to linearly scan the entire file of 100k tuples.

The false drop ratio analysis of interval signatures is the same as that for signatures obtained from text values and has been presented in Section 4. Note that for the efficiency of answering range queries with relatively large ranges, intervals in the data domain of a numerical attribute usually should not be made too small. Large intervals mean that the number of distinct interval signature is relatively small. So, a numerical attribute using interval signatures is likely to have a small effective partition size $\hat{m}$ and the effect has been analyzed in Section 4. For range queries with large ranges, the checking of many query signatures may be done parallelly.

7 Significance of M-Tree and Related Work

Though signature method has been studied extensively, the problem of translating the search result from searching signature files to the primary dataset has not received adequate attention. Usually, the cost of accessing pointers and the space overhead of pointers are ignored in the performance analyses. The connection between a signature and the data items it represents can be established by pointers. If each data item is to be pointed to by a pointer, because one signature represents many data items, the size of all pointers may well surpass the size of the signature file.

Alternatively, the primary dataset may be partitioned and only one pointer is used for each partition. We see the leaf nodes of a B+ tree naturally provide such a partitioning and a ready set of pointers. If all signatures of one attribute in an M-tree is regarded as one signature file, the B+ tree in a M-tree is a pointer structure which is shared among many multi-level signature files. This sharing of the pointer structure reduces the overall size of the entire signature-related index structure.

Besides the small size, the benefit of using M-trees lies in the fact that they have good

performance in answering conjunctive queries that involving multiple attributes. Signature method has been known for its ability to efficiently index dataset with multiple attributes by using a single superimposed signature to represent values in many attributes [10]. However, our approach is different than previous signature methods. One of the novelties of our approach is to use separate signature codings for each attribute so that for answering conjunctive queries, unqualified tuples can be pruned out early (in the high level of the tree) in the search process.

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

Signature files may be specifically partitioned to achieve faster access [8, 11, 13, 14]. However, it is the flexibility to work with arbitrary partitioning that we have explored for designing the M-tree so that it can index multiple attributes.

Multi-level signature files have been proposed previously [4, 5, 8, 15] to achieve faster access than linear scan of signature files. The uniqueness of the M-tree is that it combines together many multi-level signature files from many attributes with a B+ tree. The many multi-level signature files are not merged into one signature file. The B+ tree in the M-tree serves four purposes: it provides a partitioning 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 M-tree to achieve fast processing of conjunctive queries for tabular datasets.

There are many ways to construct multi-level signature files in the M-tree. The separate coding method [5, 15] provides an excellent performance. A separate coding file is costly to maintain if the number of levels is high [15]. 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.

8 Concluding Remarks

In this paper, we introduced the concept of the M-tree which is made by utilizing the flexibility of signature files and the efficiency of the pointer structure of a tree based index. The M-tree combines the strength of both signature files (small size) and tree based index (faster access). Because of the novel arrangement in each non-leaf node, the M-tree can

even achieve data access faster than multiple tree based indexes for queries with two or more conjunctive predicates. Analyses and experiments have been conducted to verify the performance of the M-tree.

The performance advantage and the small size of the M-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.

Some other issues were also addressed in this paper. These include using a signature file to search numerical data and performance analysis for signature method in accessing duplicate data items in the dataset. These issues have not been addressed by previous signature method research. We believe these aspects are worth exploring if the signature method is to be implemented in the mainstream database environment.

The M-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 M-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] 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.*

[11] 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.*

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

[13] 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.*

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

[15] 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.*