Improved Methods for Signature-Tree Construction

ELENI TOUSIDOU, ALEX NANOPOULOS AND YANNIS MANOLOPOULOS

Data Engineering Lab, Department of Informatics, Aristotle University, Thessaloniki, Greece 54006
Email: manolopo@delab.csd.auth.gr

Signature-based tree structures which have been proposed in the past do not perform well for large databases. The problem arises from the fact that they are incapable of pruning searching, especially at the upper tree levels, and thus they have decreased selectivities. In this paper, we locate a number of reasons for this problem and propose several methods for node splitting and partial-tree restructuring, which lead to improved query-response times. We have implemented all methods and we present experimental results, which indicate that the proposed methods are superior in all cases to the standard one and up to 5–10 times better for medium and higher weights in inclusive (partial-match) queries. Additionally, we have developed new functions for the performance estimation of signature trees which, in contrast to a previous estimation function, are able to take into account the outcome of different split methods and to provide more accurate estimation.

Received January 17, 2000; revised June 5, 2000

1. INTRODUCTION

A significant amount of today's stored data consist of records with set-valued attributes. Among others, such attributes are extensively used in object-oriented databases to represent an object's multivalued attribute, in digital library systems and search engines for the world wide web (WWW), which utilize information-retrieval methods based on set-valued attributes, in data-mining applications [1, 2] where they can represent basket market data and time-series data, and, finally, in multimedia databases and hypertext systems [3, 4] representing objects inside an image, or in www applications [5, 6] as cache content.

As an example, suppose a sample database of a company with the following class definitions, where notation '[ ]' is used for tuple constructors while '{}' is used to represent set-valued attributes:

Dept = [*dname*: str, *projs*: {**Proj**}, ...]
Proj = [*pname*: str, *emps*: {**Emp**}, *mnger*: **Emp**, ...]
Emp = [*ename*: str, *projs*: {**Proj**}, *hobbies*: {str}, ...].

The basic query for this kind of data is the *inclusive* or *partial-match* query, which retrieves all objects containing specific attributes. Inclusive queries can be also characterized as subset or superset queries, searching for all sets which are supersets or subsets of a given query set. For example, consider the query 'find all employees who like to spend their free time cooking or fishing or playing basketball', or simply

$$hobbies \supseteq \{cooking, fishing, basketball\}.$$

Set-valued attributes, such as *hobbies*, can be represented by bit vectors, called *signatures*. Each distinct element of an attribute's domain is represented by a distinct signature. The object signature is produced by superimposing each of its attribute's element signatures (this will be explained in Section 2.1). A collection of signatures comprises a signature file, which presents low space overhead and reduced update costs [7, 8]. Since signatures are abstractions of the original represented information, they introduce an information loss. As a result, non-qualifying objects, which are called *false drops*, may be retrieved as well. Several signature-extraction methods have been developed for the reduction of false-drop probability in signature files and the improvement of their retrieval performance [9, 10].

Another direction to improve the performance of signature files is the designing of tree- and hash-based signature organizations to avoid the sequential scanning of the signature file. Similar to B$^+$-trees, S-trees are height-balanced dynamic structures, which have been proposed for improving the searching performance in signature-based organizations [11]. As reported in [11], since the S-tree is based on superimposition (OR-ing) of signatures, the selectivity of the upper-level nodes tends to decrease. This affects the S-tree performance in large databases.

Other tree-based signature-file organizations include two-level [12] and multi-level [13] signature-based access methods, the latter approach being more appropriate for static data. Non-tree access methods are mostly based on the partitioning of the superimposed signature files. Horizontal or vertical fragmentation in combination with hashing techniques were studied in [14, 15, 16]. A similar approach has been investigated by Zezula *et al.*, who proposed the method of Quick Filter [17, 18].

The performance of signature-file organizations in object-oriented databases for indexing set-valued objects has been studied in [19, 20, 21], where various variations of the bit-sliced signature file were introduced. RD-trees have been proposed for indexing set-valued data and, when used with signatures, they exhibit similar performance to that of S-trees [22]. Besides inclusive queries, [23] examines the performance of signature-based structures for set-valued objects under the join query with subset/superset predicates. In [24, 25, 26], the use of signatures in path expressions has been also studied.

In the present paper, we focus on optimizing S-trees, which have been widely cited in previous works reporting performance comparisons with other structures. However, in the past, S-trees have been used under their original version, where their performance can be significantly affected by the node-splitting method. For instance, in [11] a heuristic of linear complexity is used to distribute the contents of overflowing nodes. Evidently, there does not exist an optimal solution of non-exponential complexity for the split method. For this reason, we provide several heuristics to solve this problem. Further study on the way that S-trees behave in comparison to other relevant methods can be found in [27].

More specifically, we present and evaluate a number of split methods of linear, quadratic and cubic complexity. These heuristics are based on a more careful (a) seed-selection method and (b) signature-distribution method of the node contents. Another important criterion for the S-tree performance is the selection of the leaf where a signature will be inserted. For instance, in many cases two or more paths could be followed equally well. For such cases of ties, we give a simple heuristic for this procedure, which leads to a better tree structuring. Finally, as reported in [28], the insertion order of data into a tree access method has an impact on its performance and the split method alone cannot address this problem because it takes local decisions. In the same paper, a forced reinsertion method is proposed to avoid or delay node splitting and to partially restructure the tree. Here, we adjust the reinsertion method for the S-tree, based on the properties of signature data and test its impact on S-tree performance.

We implemented all the proposed methods and carried out experimental comparison with the standard S-tree method, showing their efficiency with respect to the query performance. Additionally, we have developed several new performance-estimation functions, which provide more accurate results in comparison to a previously proposed function. Accurate estimation is important for a query processing engine. These functions are based on specific information from the resulting tree and can take into account different signature distributions resulting from different split methods.

The rest of the paper is organized as follows. Section 2 briefly gives background information on signatures and the S-tree access method. Section 3 contains a description of important performance factors and the drawbacks of the previous S-tree split method that motivated our work. Proposed methods are presented in Section 4 and performance-estimation functions are described in Section 5. Experimental results are given in Section 6, and, finally, Section 7 concludes the paper and gives directions for future work.

2. BACKGROUND

2.1. Signatures

A *signature*, symbolized by s, is a bitstring of F bits, where F is the signature's *length*. Signatures are used to indicate the presence of individuals in sets. For example, in an object-oriented database they would be used to represent a set-valued attribute of an object. Each element of a specific set can be encoded by using a hashing function into a signature, by setting exactly m out of F bits to 1, where the value of m is called the *weight* of the element signature and is often symbolized by $\gamma(s)$. The set bits are uniformly distributed in the $[1, \ldots, F]$ range, since the involved hash function is assumed to have ideal characteristics. Therefore, the probability of a bit position in an element signature to be set to 1 is equal to m/F. Finally, the set signature is generated by applying the superimposed coding technique on all element signatures, i.e. all positions are superimposed by a bitwise OR-operation to generate the set signature.

An inclusion (or subset) query searches for all objects containing certain attributes. Given an inclusion query with argument o', its query signature q is obtained by using the same methodology. The answer to the inclusion query is the collection of all objects o for which $o' \subseteq o$. If s is the signature of an object o, then it is easy to show that:

$$o' \subseteq o \Rightarrow q \subseteq s$$

where the right part of the above expression represents the fact that signature s has a 1 in all positions where the query signature also has a 1.

For example, suppose that an employee's hobbies are {basketball, cooking, hunting}, and we have to satisfy the query described in the introduction. Then the derived employee and query signatures would be as the ones illustrated in the left and right table below, respectively. As shown, since the query signature is not a subset of employee signature, the specific employee cannot be part of the answer to the query.

Element	Signature
Basketball	010001001
Cooking	000101001
Hunting	100101000
Employee	110101001

Element	Signature
Basketball	010001001
Cooking	000101001
Fishing	011001000
Query	011101001

Thus, signatures can provide a filter for testing attribute inclusion for objects, because if the subset condition does not hold for the signature, then it does not hold also for the object. Since the inverse of the above relation is not necessarily true, some objects that do not satisfy the query may be retrieved as well. These objects are called *false drops*. It has been proved that the false-drop probability is minimized when [7]:

$$F \times \ln 2 = m \times D$$

where D is the number of object attributes in a stored set.

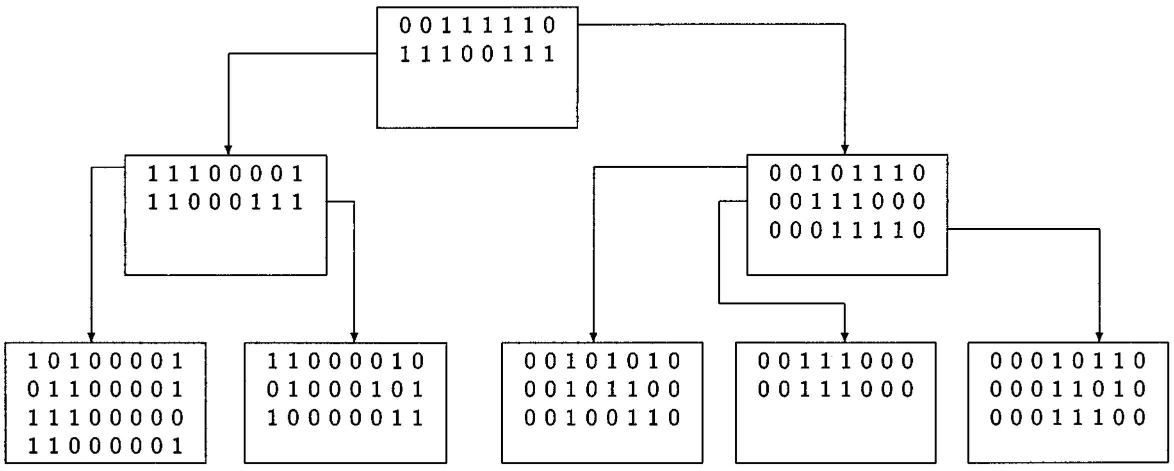


FIGURE 1. An example of an S-tree with $K = 4$ and $k = 2$.

2.2. S-trees

S-trees, similarly to B$^+$-trees, are height-balanced trees having all leaves at the same level [11]. Each node contains a number of pairs, where each pair consists of a signature and a pointer to the child node. The S-tree is defined by two integer parameters: K and k. The root can accommodate at least two and at most K pairs, whereas all other nodes can accommodate at least k and at most K pairs. Unlike B-trees where $k = K/2$, here it holds that: $1 \le k \le K/2$. The tree height for n signatures is at most: $h = \lceil \log_k n - 1 \rceil$. Signatures in internal nodes are formed by superimposing (OR-ing) the signatures of their children nodes.

Due to the hashing technique used to extract the object signatures, the S-tree may contain duplicate signatures corresponding to different objects. In Figure 1, an example of an S-tree with height $h = 3$ is depicted, where signatures in the leaves represent individual set signatures (i.e. the indexed objects). For simplicity these signatures are assumed to be of equal weight, i.e. $\gamma(s) = 3$, but they vary from 3 to 6 in upper levels due to superimposition.

Successful searches in an S-tree proceed as follows. Given a user query for all sets that contain a specified subset of objects, we compute its signature and compare it to the signatures stored in the root. For all signatures of the root that contain 1s at least at the same positions as the query signature, we follow the pointers to the children of the root. Evidently, more than one signature may satisfy this comparison. The process is repeated recursively for all these children down to the leaf level following multiple paths. Thus, at the leaf level, all signatures satisfying the user query lead to the objects that may be the desired ones (after discarding false drops). In the case of an unsuccessful search, searching may stop early at some level above the leaf level, if the query signature has 1s at positions where the stored signatures have 0s.

For the insertion of a new signature, the appropriate leaf is selected by traversing the tree top-down and choosing at each level the child node whose signature will require the minimum weight increase. If s' is the new signature and s is the signature of a node, then the weight increase ϵ is [11]:

$$\epsilon = \gamma(s \vee s') - \gamma(s).$$

Thus, selecting the node with minimum ϵ aims at the minimization of the number of multiple paths that have to be followed. As a tie criterion, the node with minimum Hamming distance may be used. After the insertion in the leaf, the parent-node signature may need to be updated. Therefore, the update of signatures at ancestor nodes may propagate up to the root.

If the leaf, where a new signature is to be inserted, is already full, i.e. it contains K entries, then it is split. A new node is created and the $K + 1$ entries have to be distributed between the two nodes so that the probability of accessing both nodes together (i.e. by the same query) is as low as possible. The splitting algorithm described in [11] can be viewed as consisting of two phases: the *seed-selection phase* and the *signature-distribution phase*. During the seed-selection phase, we locate (a) the signature with the highest weight, called seed α, and (b) the signature with the maximum weight increase ϵ from seed α, which is called seed β. Seed α and seed β are assigned to the two leaves that result from the split. During the signature-distribution phase, the remaining signatures are considered one-by-one, with no particular ordering, and assigned to one of the two pages. More specifically, every signature is superimposed with both seeds, the weight increases are calculated, and then it is stored in the node of the seed for which the weight increase is smaller. Ties are resolved by taking the minimum Hamming-distance criterion. When one of the nodes has already accepted $K - k + 1$ entries, then the remaining entries are forced to be assigned to the other node so that the *minimum-containment criterion* is fulfilled, without taking into account the weight-increase criterion. It is easy to prove that this method has linear complexity.

Due to the superimposition technique, nodes near the root tend to have heavy signatures (i.e. with many 1s) and thus they have low selectivity. For such a case, it has been proposed to cut off the top tree levels and to form a forest of a certain number of independent S-trees [11], which all need to be searched upon a query. Although this action may reduce the accesses to internal nodes for low-weight queries, it does not resolve the problem of excess disk accesses to the leaf nodes. Since the number of leaf nodes is much larger than the number of internal nodes, the overall performance of S-trees is affected by the disk accesses to the leaf nodes. The following sections present several improved methods for the S-tree construction to address the problem without the need to cut off the tree.

3. PERFORMANCE FACTORS AND MOTIVATION

Several factors affect the performance behavior of S-trees during query execution. Since S-trees resemble R-trees, multiple paths can be followed to answer an inclusion query. The reduction of I/O complexity during query execution can be achieved by the following.

Weight minimization at each node. Weight minimization results in fewer 1s in each node so that the probability of a node being invoked during an inclusion query is reduced.

Overlap minimization between nodes. Overlapping between any two nodes (co-occurrence of 1s at the same positions) should also be decreased, since this reduces the number of multiple paths to be followed.

Storage utilization maximization. Storage utilization is tuned by parameter k. Larger k values result in trees with fewer nodes. Such small trees are more suitable for inclusion queries with low weight, since such queries invoke a large tree portion. (In this case the previous two factors are not as important.)

It is easy to notice that the above factors are contradictory. Weight minimization may result in more overlap between nodes and vice versa. Additionally, weight and overlap minimization may require lower k values, which reduces storage utilization.

S-trees try to minimize each node's weight during insertions or node splits, whereas they do not pay attention to the rest of the factors. Since S-trees have many similarities with R-tree-like access methods [29], performance tuning of S-trees can be achieved with approaches followed by existing spatial access methods.

There is an analogy between weight minimization in S-trees and area minimization in R-tree nodes. It has been noticed in [28, 30] for R-trees (and it also holds for S-trees as verified by our experiments) that the insertion of a new entry is biased towards the node with more entries. This takes place during the invocation of the *choose-leaf* procedure, and can be explained by the fact that the node with more entries has already high weight (large area in R-trees) and it requires the least weight increase (enlargement in R-trees).

The same applies for the *split* procedure, where the initial seeds are the entry α with the highest weight and the entry β with the maximum weight increase when superimposed with α. The assignment of the remaining entries tends to prefer the node containing entry α, because it will require the least weight increase due to its high weight. Moreover, after the first few assignments, this problem is escalated since the node containing α is continuing to accumulate more entries. Also, the minimum-containment criterion is another source of complexity. These facts result in an uneven distribution of entries between the two nodes, which affects performance and storage utilization.

Finally, since S-trees (as any other tree-structured access method) depend on the insertion order, the *forced reinsertion* technique, which has been proposed in [28], can be applied to achieve efficient dynamic tree reorganization.

Although there are analogies between R-tree-like access methods and S-trees, approaches from the spatial domain cannot be followed directly since they exploit several geometric properties of the entries. In the following, we will present four splitting algorithms, one of linear, two of quadratic, and one of cubic complexity. Our linear algorithm behaves similarly to the original linear algorithm [11]. The first quadratic algorithm adapts the R-tree-splitting algorithm, taking into account signature properties. The other new quadratic split algorithm is based on hierarchical clustering. Finally, the cubic algorithm is based on an exhaustive search of the best couple of seeds, based on a minimization criterion. Each method tries to optimize one or more of the previously mentioned factors. Additionally, we present the application of the forced reinsertion method to S-trees.

4. PROPOSED METHODS

4.1. A refined linear split

The first variation focuses on the seed selection phase and adapts a technique from [31]. In particular, our variation of the original linear split method is based on a different choice of the two seeds that have to be distant according to the weight criterion. Instead of choosing the first pair of seeds that come out as a result of the linear algorithm, we could continue searching for such pairs, m times. Specifically, after randomly choosing one of the signatures as the starting α seed, we search for the β seed and continue by replacing the α seed with the β one. Then we start searching for a new β seed and we keep on until we find m different pairs. We keep the last one as the pair of our choice. The algorithm of linear distribution is then followed to fill the two produced nodes.

```
Algorithm: Refined choice of seeds $\alpha$ and $\beta$ after $m$
           repetitions.
           1. Let seed $\beta$ be a randomly chosen
              signature of the set $S$.
           2. $i \leftarrow 0$.
Loop:      3. If $i = m$, then stop.
           4. seed $\alpha \leftarrow$ seed $\beta$.
           5. Find seed $\beta$ according to the original
              linear splitting algorithm.
           6. $i \leftarrow i + 1$.
           7. Go to Loop.
```

This method tries to minimize the overlap between the two new nodes after the split by selecting two seeds, which are as far apart as possible with respect to the weight-increase criterion. It should be noticed that it does not choose the heaviest signature as the starting seed in order to avoid any bias that could be created. Knowing that the furthest pair can only be found with quadratic complexity and since this algorithm is linear on the number of repetitions, it can only find an approximation of it.

4.2. Quadratic split

The second variation focuses on the signature distribution phase. In particular, we retain the original algorithm for choosing seeds but we change the way we distribute the remaining signatures to the two nodes. This new approach is based on the R-tree split algorithm, which involves quadratic complexity [32]. In other words, after assigning the two seeds into the respective nodes, we search for the entry with the maximum difference of the weight increase in the two nodes and insert it in the appropriate one. We continue this way until one of the nodes is filled with $K - k + 1$ entries. In the following, the quadratic split algorithm is described.

```
Algorithm: Quadratic split of a node's signatures $S$
           into nodes nodeA and nodeB.
           1. Choose seeds $\alpha$ and $\beta$.
           2. $signA \leftarrow \alpha$ and $signB \leftarrow \beta$.
Loop:      3. For each $s_i \in S$ calculate the weight
              increases $\epsilon(A) = \gamma(signA \vee s) - \gamma(signA)$
              and $\epsilon(B) = \gamma(signB \vee s) - \gamma(signB)$.
           4. Choose the signature $s_i$ with the
              maximum $|\epsilon(A) - \epsilon(B)|$.
           5. If $\epsilon(A) < \epsilon(B)$, then insert $s_i$ in nodeA
              and calculate $signA \leftarrow signA \vee s_i$,
              else if $\epsilon(A) > \epsilon(B)$, then insert $s_i$ in
              nodeB and calculate
              $signB \leftarrow signB \vee s_i$,
              else insert in the node with the fewer
              entries and perform the appropriate
              superimposition.
           6. $S \leftarrow S - \{s_i\}$.
           7. Go to Loop.
```

Although this method presents an increase in time complexity, it performs a more careful signature assignment to nodes compared to the linear assignment method, resulting in decreased node weights.

4.3. Cubic split

The third variation also focuses on the seed selection phase and adapts a technique from [33].

```
Algorithm: Exhaustive search for seeds $\alpha$ and $\beta$,
           followed by a linear distribution.
           1. minMaxWeight $\leftarrow$ MAX INTEGER.
Loop:      2. For each pair of signatures, $sign(i)$ and
              $sign(j)$, apply the original linear
              split algorithm and distribute all
              entries between nodeA and nodeB.
           3. In each iteration, calculate the weights
              of the superimposed signatures that are
              produced by each one of the two nodes.
              Let maxWeight be the heavier of
              the two.
           4. If $maxWeight < minMaxWeight$, then
              $minMaxWeight \leftarrow maxWeight$,
              $\alpha \leftarrow sign(i)$ and $\beta \leftarrow sign(j)$.
           5. Go to Loop.
```

In this method, we perform an exhaustive search for the best couple of seeds that should be chosen. As a metric, we consider the weight of the superimposed signatures that result after the split. More specifically, we begin by applying on every possible couple of seeds the distribution of the remaining entries, as indicated by the original algorithm of linear split. For each such couple, we superimpose the signatures that resulted in each one of the two new nodes, *nodeA* and *nodeB*, and measure the weights of the two produced signatures α and β. By taking into account only the heavier of the two, we will finally select the pair of seeds that produces the minimum of these heaviest weights (MinMax criterion).

This method requires all possible pairs to be tested as possible seeds. For each pair, a linear assignment of the remaining entries is performed. Therefore, the time complexity of the method is cubic. The method is trying to minimize both the overlapping (by selecting appropriate seeds) and the weights in the resulting two nodes. Notice that for each pair, the distribution of the remaining entries could be performed with quadratic complexity, as in the previous method. This would require an $O(n^4)$ time complexity (n is the number of signatures in the node), which would heavily affect the performance of the access method during insertions. For this reason we did not test methods of complexity higher than cubic.

4.4. Hierarchical clustering

The last one of the methods considered was based on the hierarchical clustering method, since the notion of splitting a node can be considered similar to that of clustering samples into two clusters corresponding to the two empty nodes. Let us consider a sequence of partitions of the n signatures, where the first partition comprises n clusters, each cluster containing exactly one sample. The next is a partition of $n - 1$ clusters, the next a partition of $n - 2$, and so on until the nth partition, in which all samples form one cluster. If the sequence has the property that whenever two samples are in the same cluster at level k they also remain together at all higher levels, then the sequence is said to be a *hierarchical clustering*. In the literature, two classes of hierarchical clustering have been reported. The *agglomerative* (bottom-up, clumping) one, which starts with n singleton clusters and forms the sequence by successively merging clusters, and the *divisive* (top-down, splitting) class that starts with all samples in one cluster and forms the sequence by successively splitting clusters.

Transferring this problem of clustering in the signature domain, the procedure that best fits this domain is the agglomerative one, since it will make a more refined split of the set of signatures. For our purposes, the merging sequence will stop at the $(n-1)$th partition, when two clusters will be left.

Next, the respective algorithm is shown, where s_i is one of the node signatures and C_i is a created cluster.

```
Algorithm: Basic agglomerative clustering.
           1. $k \leftarrow n$.
           2. $C_i = \{s_i\}$ for $i = 1, \ldots, n$.
Loop:      3. If $k \leq 2$, then stop.
           4. Find the nearest pair of distinct clusters,
              say $C_i$ and $C_j$.
           5. Merge $C_i$ and $C_j$, delete $C_j$.
           6. $k \leftarrow k - 1$.
           7. Go to Loop.
```

In order to decide which two clusters will be merged in each step, a distance function has to be defined. After testing several distance measures, we decided to restrict our attention to the following:

$$d_{\min}(C_i, C_j) = \min_{s \epsilon C_i, s' \epsilon C_j} \mathit{hammDist}(s, s')$$

$$d_{\mathrm{ave}}(C_i, C_j) = \|x_i - x_j\|$$

where function *hammDist* is the Hamming distance between the two signatures. In the first case, we search for the pair of clusters whose corresponding superimposed signatures present the minimum Hamming distance. This way, in every step the most similar clusters will be joined together. In the second case, we search for the pair of clusters that shows the minimum Euclidean distance among signatures. More specifically, for each created cluster, an array can be produced with size equal to the signature length. Each entry in the array will contain the mean value of the signatures' weight in this position. The goal is to find the two clusters whose respective arrays show the minimum Euclidean distance.

Suppose that two clusters A and B are given, with three signatures each as follows:

Cluster A	Cluster B
11000010	00101010
01000101	00101100
10000011	00100110

The produced arrays for these two clusters are:

$$x_{\mathrm{A}} = (2/3, 2/3, 0, 0, 0, 1/3, 2/3, 2/3)$$
$$x_{\mathrm{B}} = (0, 0, 1, 0, 2/3, 2/3, 2/3, 0).$$

Thus, the Euclidean distance between Cluster A and Cluster B is:

$$\begin{aligned} d_{\mathrm{ave}}^2 &= (2/3-0)^2 + (2/3-0)^2 + (0-1)^2 + (0-0)^2 \\ &\quad + (0-2/3)^2 + (1/3-2/3)^2 + (2/3-2/3)^2 \\ &\quad + (2/3-0)^2 = 25/9. \end{aligned}$$

The objective of a clustering method is to minimize the distance for all objects inside a cluster and to maximize the distance of objects of different clusters. Therefore, the split based on hierarchical clustering tries both to minimize the overlap of the two nodes and the weight of each one. As proved in [34], the complexity of the method is quadratic.

4.5. Other examined heuristics

All split methods described previously try to minimize either the weight in each node, or the overlapping between the nodes, or both criteria. With respect to the third factor, i.e. storage utilization maximization, all algorithms have to satisfy the condition that each node contains at least a minimum number of entries. This is done during the signature-distribution phase for all algorithms. When necessary, all remaining entries are assigned to one of the nodes so that it will contain the minimum number of entries. Evidently, this may reduce the method effectiveness with respect to the other two factors but, as has been mentioned, it is important for queries with low weights.

From the experiments performed we realized that a number of tie-break criteria have to be applied to increase the performance of the tested methods. During top-down traversing or during the split process, when the weight increase is equal for two nodes, we use the Hamming-distance criterion to solve ties. In the case that the tie still holds, we choose the node with the fewer entries so that both nodes will present a minimum occupancy. It has to be noticed that the application of tie-break criteria is important because the *split* and *choose-leaf* procedures are resolved by them very often. This is not usual in spatial access methods and this is due to the distribution of 1s at the upper tree levels.

Apart from the previous methods, we experimented with a number of other splitting variations, with no significant improvement over the original linear split algorithm. In short, some of the tested algorithms were the following.

4.5.1. Seed selection variations

- We select as seed β the signature maximizing the Hamming distance and not the signature maximizing the weight increase.
- We find seed α and seed β as in the original algorithm. We insert seed β in the new node and keep on searching for other β seeds from the overflowed node to store them in the new node until the latter node reaches the minimum occupancy.

4.5.2. Signature distribution variations

- We sort the remaining signatures in decreasing order, based on the difference of the weight increase that they would cause if inserted in each node. We choose the one with the largest difference and insert it in the node where it causes the least increase. From this point on, the algorithm may be either linear or quadratic. In the first case, we could continue with the second in the list and insert it in the appropriate node as previously, whereas in the second case we could perform the sorting algorithm in the remaining entries (since one of the nodes has changed), and continue accordingly.

- We make two sorted lists in increasing order. In the first (second) list we store the signatures that increase less the weight of the first node than of the second (of the second node than of the first one, respectively). At each iteration we choose the signature with the lesser increase in one of the two nodes and insert it in the appropriate node. As before, the algorithm may have a linear or quadratic extension depending on whether we repeat the sorting algorithm or not.
- We followed the *inner-product method*, which has been proposed as a means to decluster the S-tree pages in parallel disks [35]. Our algorithm was based on the inner product of the signatures found in each page and was applied on the way the signatures of each overflowed node were distributed in the two new nodes. More details about this method can be found in [35].

4.6. Forced reinsertion

When Beckmann *et al.* introduced the R*-tree in [28], they also proposed the technique of forcing the reinsertion of part of the overflowed tree node. Forcing the reinsertion of some entries could achieve a better clustering of signatures in pages, by replacing older entries, which are no more appropriate for a specific page. Apparently, to some extent clustering is accomplished during the split process. In R*-trees and similarly in S-trees, the reinsertion technique used is the following. Whenever an overflow occurs, we delay the split process, or even avoid it, by eliminating a number of the entries from the node and trying to reinsert them always in the level from which they were eliminated. As shown in [28], forced reinsertion can yield an improvement of up to 50% in retrieval performance. In the sequel, we only give the algorithm of reinsertion adapted from the respective R*-tree algorithm, so that it matches the S-tree needs. More details for the rest of the algorithm can be found in [28].

```
Algorithm: Reinsertion technique.
           1. Let $s$ the heaviest signature of all $K + 1$
              entries of node $N$.
           2. counter ← 0.
           3. signA ← 0.
Loop:      4. signA ← signA ∨ $s$.
           5. Compute the weight increase that could
              be caused in signA by each of the
              remaining entries.
           6. Let $s$ be the one causing the greatest
              increase, remove it from node $N$, insert
              it in a queue and increase counter.
           7. If counter = $p$, then go to step 8, else go
              to Loop.
           8. Invoke Insert for items in the queue, to
              reinsert the entries.
```

Apart from the above algorithm, we experimented with a variation, where the choice of signatures to reinsert is treated as a virtual split. More specifically, we perform a quadratic split (as described in Section 4.2), where the new (virtual) node is filled to the reinsertion threshold p. The signatures of the latter node will be reinserted, whereas the other node will play the role of the overflowed one. The quadratic split was chosen because, this way, the inherent complexity of reinsertion will remain unaffected. We will see how this variation affects the performance of the methods in Section 6.3, where we refer to the first kind of reinsertion as the original method (using as notation the suffix _RE), while we refer to the second as the quadratic one (using the suffix _QRE, due to the virtual quadratic split).

It is noted that for all the previous split methods, whenever the insertion is completed at the leaf level, all parent signatures in the insertion path have to be updated by using the technique of superimposed coding on their children signatures.

5. PERFORMANCE ESTIMATION

The response time for a partial match query is the time required to retrieve all corresponding pages from disk plus the time to process their contents. The processing involves fast bit-manipulation methods and can be considered as negligible. The performance measure in the rest of the section is the number of disk accesses. Additionally, we assume that all signatures have the same weight γ and that in each signature all 1s are uniformly distributed. The same assumption for uniform distribution of 1s holds also for the query signatures. These are realistic assumptions for many hash functions for signature generation. Table 1 gives all symbols used in the following.

The estimation of disk accesses for a partial match query with weight γ_q can be made independently of the resulting tree. This approach is followed in [11], where it is shown that:

$$p(\gamma_q, d) = \left[1 - \left(1 - \frac{\gamma}{F}\right)^{\lambda(d)}\right]^{\gamma_q} \tag{1}$$

is the probability of a signature at depth d (where $1 \leq d \leq h$ and h is the tree height) to contain 1s at γ_q prespecified positions. Symbol $\lambda(d)$ denotes the number of signatures at the leaf level, which belong to a subtree rooted at a node at depth d and is:

$$\lambda(d) = \frac{n}{\prod_{i=1}^{d} \overline{n_i}} \tag{2}$$

where $\overline{n_i}$ is the average number of signatures per node at depth i.

The expected number of disk accesses is the sum of the number of expected disk accesses at each depth plus one disk access for the root:

$$DA = \sum_{d=1}^{h-1} \prod_{i=1}^{d} \overline{n_i}\, p(\gamma_q, i) + 1. \tag{3}$$

The above estimation is based on the assumption [11] that for signatures at the upper levels, all F positions are

TABLE 1. Symbol table.

Symbol	Definition
h	Tree height
n	Total number of signatures
$\overline{n_d}$	Average number of a node's signatures at depth d ($1 \leq d \leq h$)
N	Total number of tree nodes
γ_i	Weight of the signature of node i (in father node)
F	Signature length
γ	Signature weight
γ_q	Query-signature weight
$\overline{\gamma_d}$	Average signature weight at depth d ($1 \leq d \leq h$)
P	Number of histogram partitions
P_i	Number of weight values in i-th histogram partition ($1 \leq i \leq P$)
$\overline{\gamma_i}$	Average weight within i-th histogram partition ($1 \leq i \leq P$)

set with equal probability. Different split methods result in different distributions of 1s at the upper levels, not necessarily uniform, so this assumption does not hold in general. Information about the distribution of 1s can be taken into account by considering the weights of tree nodes after the tree construction. Given a node i whose covering signature (superimposed signature of all signatures of node i) in its father node has weight γ_i, the following lemma gives the probability that node i will be invoked during a partial match query.

LEMMA 1. *The probability that a tree node i with covering signature of weight γ_i will be fetched by a partial match query of weight γ_q is:*

$$p(\gamma_q, \gamma_i) = \frac{\binom{\gamma_i}{\gamma_q}}{\binom{F}{\gamma_q}}. \tag{4}$$

Proof. Each of the $\binom{\gamma_i}{\gamma_q}$ combinations is equi-probable (uniformity assumption for query signatures) and corresponds to a partial match query with weight γ_q, which will require the retrieval of node i. The total number of all possible partial match queries of weight γ_q is $\binom{F}{\gamma_q}$. It simply follows that the fraction of queries, which will retrieve node i over all possible queries, gives the probability that node i will be fetched. □

The root does not have a corresponding father signature and its probability is one. Lemma 2 gives an estimation of the total disk accesses.

LEMMA 2. *The expected number of total disk accesses for a partial match query of weight γ_q is:*

$$DA = \sum_{i=1}^{N} p(\gamma_q, \gamma_i) = \sum_{i=2}^{N} \frac{\binom{\gamma_i}{\gamma_q}}{\binom{F}{\gamma_q}} + 1. \tag{5}$$

Proof. From Lemma 1, every node i contributes one disk access with probability $p(\gamma_q, \gamma_i)$. Therefore, the expected number of disk accesses is given by the sum of probabilities for all tree nodes besides the root. The root contributes one additional disk access, i.e. it is fetched with probability one. □

With simple algebraic manipulations the above formula can be expressed as:

$$DA = \frac{1}{\prod_{k=0}^{\gamma_q-1}(F-k)} \sum_{i=1}^{N} \left(\prod_{k=0}^{\gamma_q-1} (\gamma_i - k) \right) + 1 \tag{6}$$

from which it follows that the number of disk accesses is heavily related to the signature weights, especially at the upper tree levels where the weights are larger. Each split policy results in a different distribution of weights and the above estimation of the number of disk accesses considers the outcome of each split method.

The estimation of Equation (5) requires information for every tree node. This presents an O(N) space and O($N\gamma_q$) time complexity. A more simple measure can use the average weight $\overline{\gamma_i}$ for each tree depth. Since each tree depth d has $\prod_{i=1}^{d} \overline{n_i}$ signatures the expected total number of total disk accesses is:

$$DA = \frac{1}{\binom{F}{\gamma_q}} \sum_{d=1}^{h-1} \left(\binom{\overline{\gamma_d}}{\gamma_q} \prod_{i=1}^{d} \overline{n_i} \right) + 1 \tag{7}$$

which is easily derived from Equation (5), considering all signatures at depth d to be of weight $\overline{\gamma_d}$. The required space complexity is reduced to O($h-1$), where h is the tree height, with O($h!$) time complexity (in our experiments: $h \leq 5$).

Although Equation (7) provides a simple and computationally cheap estimation of the number of disk accesses, it is based on the approximation of all signature weights of a tree depth with their average value. The signature weights for a tree depth may vary significantly, so the approximation of Equation (7) will provide inaccurate estimations.

A computationally non-expensive estimation, which considers the distribution of signature weights at the same time, can be based on histograms for an approximation of the weight distribution. The signature weights are partitioned into P ranges of values, each one containing P_i weights. The partitioning is done into equi-width ranges. The weights

TABLE 2. Parameters used in experiments and the values tested.

Parameter	Values
Number of inserted signatures ($\times 10^3$)	$N = 10, 50, 100, 150$
Signature size/weight (in bits)	$F/\gamma = 512/80, 512/120, 1024/120, 1024/256$
Page size (in KB)	1, 2, 4
Minimum page capacity (as percentage)	$k = 35\%$ of maximum page capacity

of each partition are represented with their average value $\overline{\gamma_i}$ within the partition. Following this approximation, the expected number of disk accesses is:

$$DA = \sum_{i=1}^{P} \frac{\binom{\overline{\gamma_f}}{\gamma_q} P_i}{\binom{F}{\gamma_q}} + 1. \qquad (8)$$

The estimation of Equation (8) presents O(P) space complexity and O($P\gamma_q$) time complexity. Since $P \ll N$, Equation (8) is not as computationally expensive as Equation (5) and is more accurate than Equation (7), since it provides a better approximation of the weight distribution. The histogram can be stored along with the S-tree and can be dynamically updated after each insertion, or after a prespecified number of insertions. This does not present a significant overhead to the insertion algorithm.

6. EXPERIMENTAL RESULTS

All previously described splitting algorithms were tested experimentally under varying parameters to evaluate their performance. The structures were implemented in C++ and the experiments run on a Pentium II workstation under Windows NT. Along the lines of the experimentation by Deppisch [11], the considered parameters and the tested values for each parameter that we used are given in Table 2.

For each experiment, we created signatures randomly using a uniform distribution for the positions that will be set to 1. The performance measure was considered to be the number of disk accesses required to satisfy a query. For each query weight, an average of 100 measurements was taken.

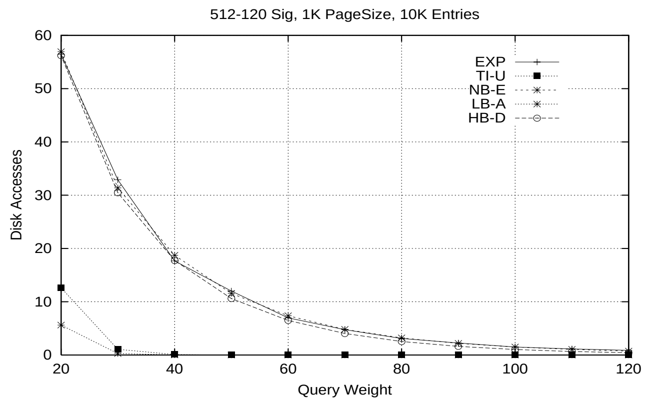
6.1. Evaluation of estimation functions

First we evaluated the accuracy of the estimation functions given in Section 5. Estimates for each query weight were compared with the results obtained by performing the queries of the same weight to the actual implementation of the corresponding S-tree.

The acronyms that are used in the graphs giving the result of comparison are:

Exp for the actual experimental result,

TI-U (Tree-Independent Uniform) for the estimation of Equation (3), which is independent of the underlying tree and is based on the assumption that 1s in signatures of the upper levels are uniformly distributed,



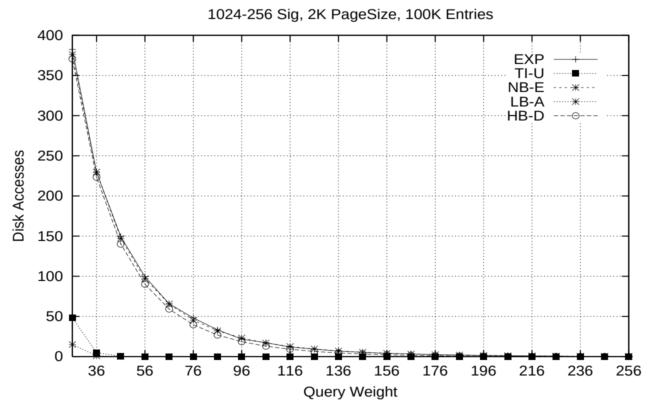


FIGURE 2. Comparison of estimation functions of the original (upper) and quadratic (lower) split methods as a function of the query weight.

NB-E (Node-Based Exhaustive) for the estimation of Equation (5), which is based on weights obtained from each node,

LB-A (Level-Based Average) for the estimation of Equation (7), which is based on the average weight for each tree level, and

HB-D (Histogram-Based Distribution) for the estimation of Equation (8), which is based on the distribution of signature weights obtained from a histogram.

Figure 2, in the upper part, illustrates the results for 10,000 entries of length 512 and weight 120. The used split algorithm was the original linear one. In the lower part, results are given for 100,000 entries of length 1024 and weight 256, where the split algorithm was the quadratic. As shown, TI-U cannot provide accurate results because

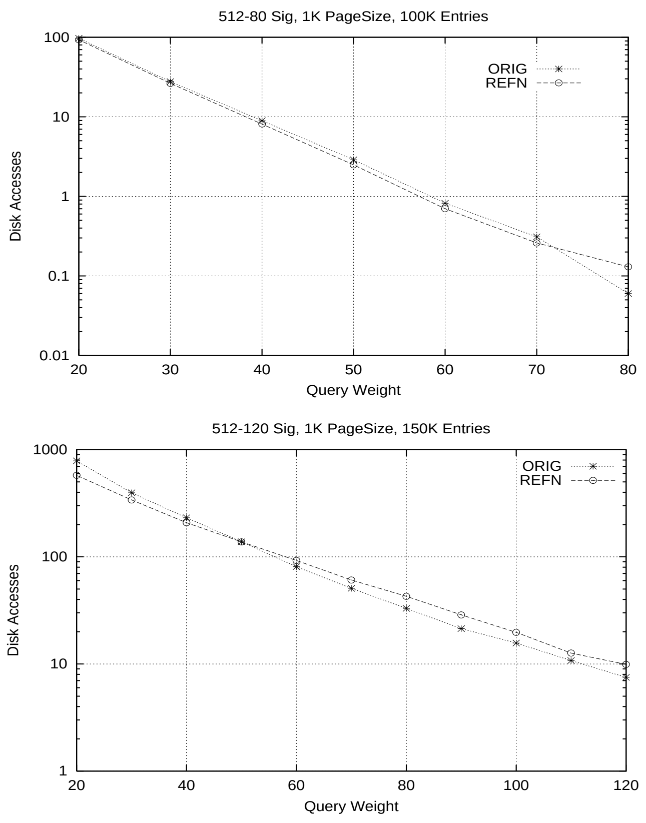


FIGURE 3. Performance of the two linear algorithms as a function of the query weight.

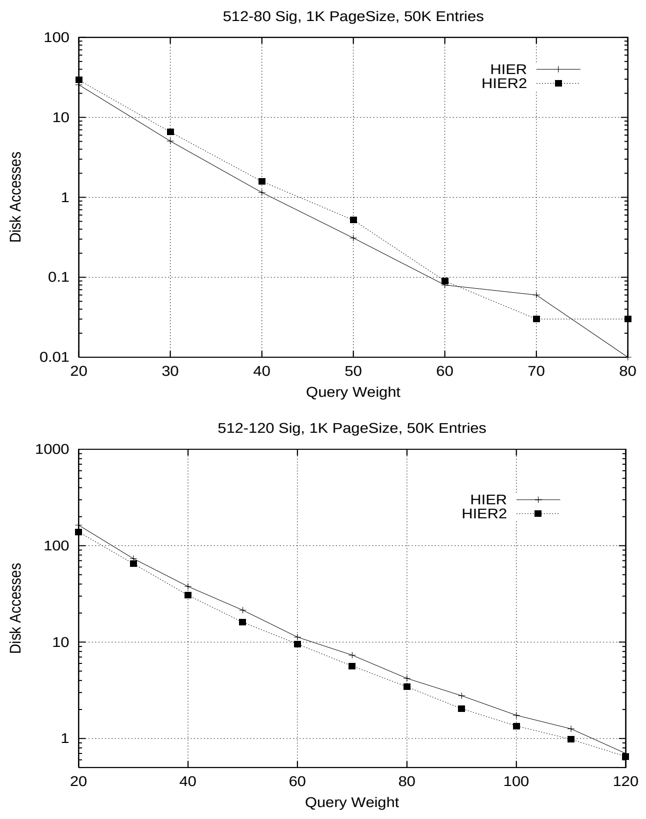


FIGURE 4. Performance of the two variations of hierarchical clustering (minimum and mean distance) as a function of the query weight.

the assumption that the distribution of 1s at the upper tree levels is uniform, is not adequate. Also, LB-A which is based on the approximation of the query weight for each tree level with the average value of weight for that level, does not provide accurate estimations. LB-A tries to overcome the assumption of uniformity made by TI-U but the use of average weights only does not capture the distribution of weight at the upper tree levels. On the other hand, NB-E provides an accurate estimation. The error of NB-E compared to the experimental results (Exp) is not larger than 3%. Also, HB-D gives accurate estimates with an error between 2 and 15%. Evidently, the improved accuracy of NB-E is achieved at the cost of increased computation and space complexity. Therefore, in our experiments HB-D provides results which are close to the ones of NB-E, while being less computationally expensive. The same conclusions were derived for all other split algorithms and for different parameter values.

6.2. Evaluation of split algorithms

The acronyms used in the graphs where the results are represented are: ORIG for original linear split, REFN for refined linear split, CBIC for cubic split, QUAD for quadratic split, HIER for hierarchical clustering with minimum distance, and HIER2 for hierarchical clustering with mean distance.

As far as the linear algorithms are concerned, the choice of seeds by itself does not seem to have any effect on the general performance of the splitting procedure. This can be understood by the performance of the refined original linear split as shown in Figure 3. By trying to find the best couple of seeds after $m = 5$ times and then applying the linear signature distribution, this technique did not perform significantly better than the original algorithm, while sometimes it performed slightly worse.

Regarding the two functions of hierarchical clustering as shown in Figure 4, when the minimum distance heuristic is applied (HIER) the method excels in the lightest cases (i.e. 512-80 and 1024-120 signatures), whereas for the rest of the cases the mean-distance heuristic is more efficient. Due to the very similar performance of the two functions, and in order to present more ‘readable’ graphs, we will only show hierarchical clustering with mean distance as a representative of the two methods.

In the upper part of Figures 5 and 6 we present the disk accesses with respect to the query signature weight for the four most interesting methods (original linear, quadratic, hierarchical clustering with mean distance and cubic) for two representative signature sizes (i.e. 512-120 and 1024-256). In the lower part of Figures 5 and 6 the normalized results are shown with respect to the cubic method. The original linear split method (ORIG) presents the worst performance in all cases. The cubic split method (CBIC) clearly outperforms all other methods. As far as the

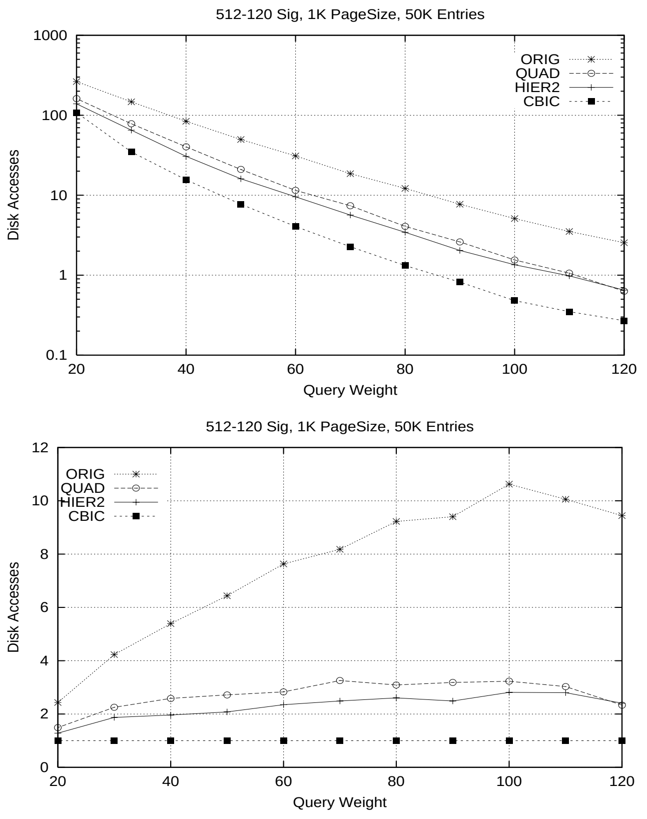


FIGURE 5. Performance of the proposed methods for 512-120 signatures (normalized results in the lower part) as a function of the query weight.

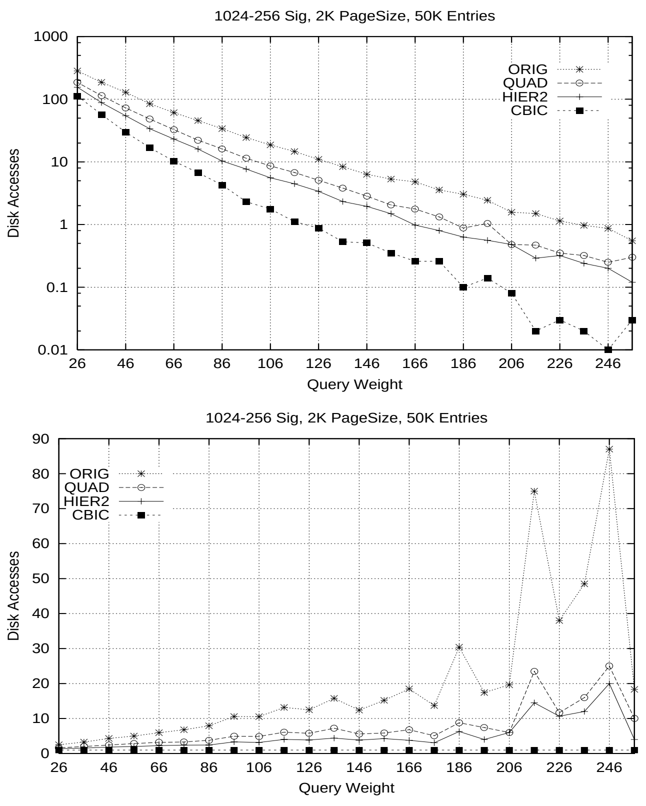


FIGURE 6. Performance of the proposed methods for 1024-256 signatures (normalized results in the lower part) as a function of the query weight.

quadratic complexity algorithms are concerned, both QUAD and HIER2 perform pretty close together, with HIER2 performing slightly better. The exhaustive searching in cubic split for the two seeds seems to play a significant role in combination with the creation of nodes with the smallest possible superimposed weights, needed to reduce as much as possible the concentration of 1s. Split methods of quadratic and cubic complexities present an overhead to the insertion time but this pays off during query processing.

Figures 7 and 8 (in their upper parts) illustrate the disk accesses of the same methods with respect to the number of signatures that are stored in the tree. Again the same two representative signature sizes were chosen. The lower parts of Figures 7 and 8 present the normalized results with respect to the cubic method. It can be easily seen that all methods present a linear behavior, where the original linear split method has the largest increase with respect to the number of disk accesses. Again, the cubic split method outperforms all other methods, whereas the HIER2 method presents the second-best performance. Both quadratic and cubic split methods scale much better compared to the original linear method and can be used for large signature databases.

Figure 9 presents the disk accesses with respect to the query signature weight, using double page sizes. By doubling the space size it can be clearly seen that the performance deteriorates greatly, no matter what split

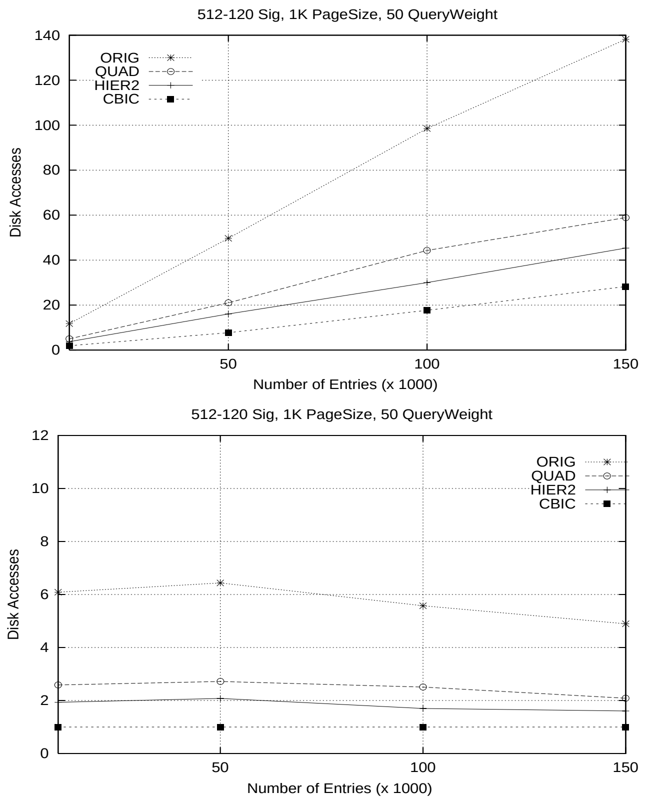
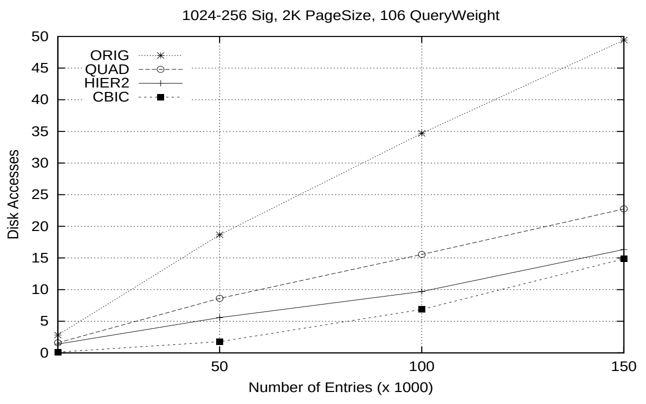


FIGURE 7. Performance of the proposed methods for 512-120 signatures (normalized results in the lower part) as a function of the number of signatures.



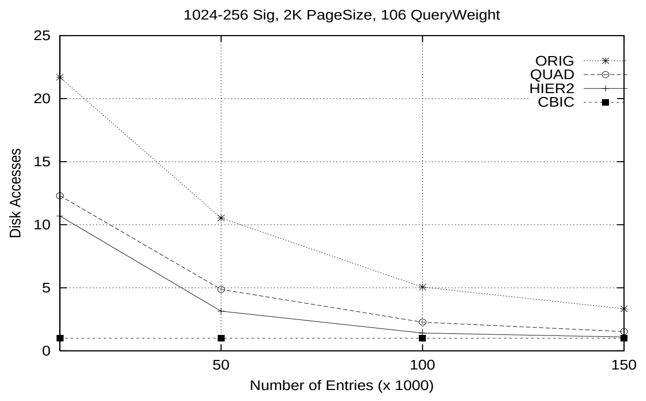
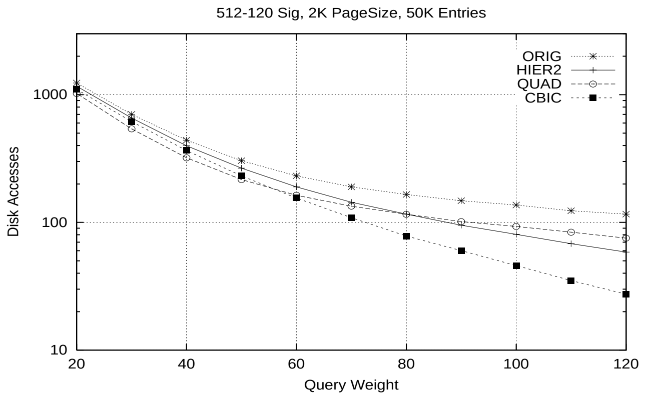


FIGURE 8. Performance of the proposed methods for 1024-256 signatures (normalized results in the lower part) as a function of the number of signatures.



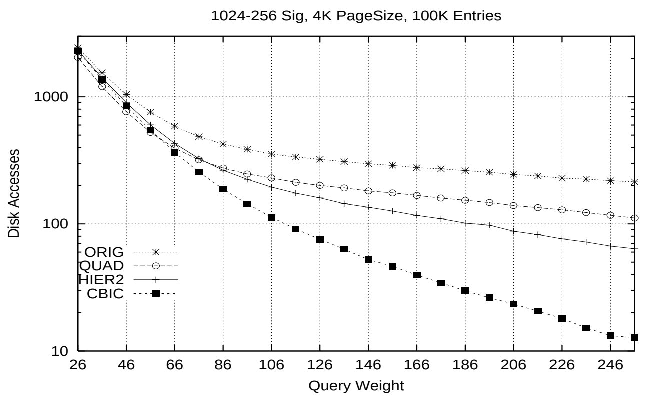
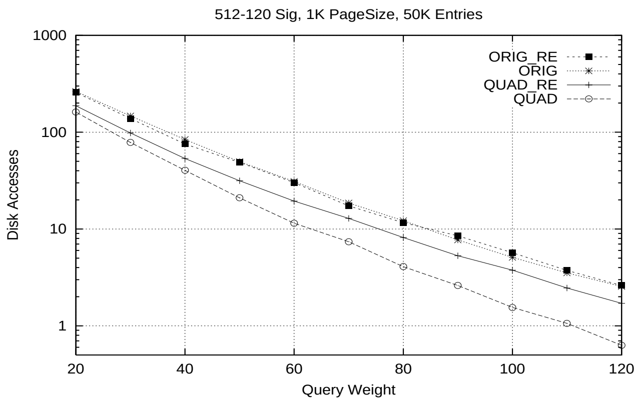


FIGURE 9. Performance of the proposed methods for larger page sizes. Top: 2K-512 signatures. Bottom: 4K-1024 signatures.



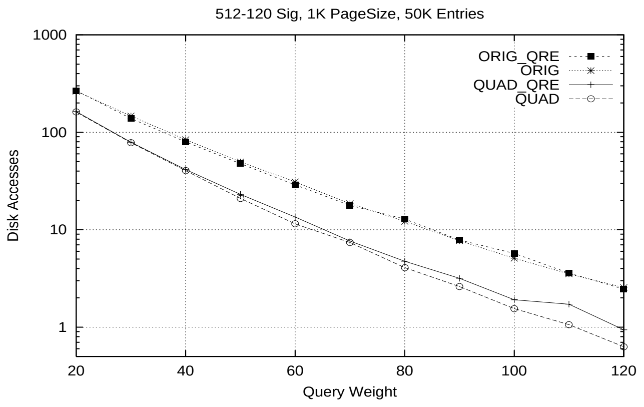


FIGURE 10. Performance of the reinsertion technique when applied on the original and quadratic split methods. Top: original reinsertion with 30% threshold. Bottom: quadratic reinsertion with 15% threshold.

method has been used. This is the consequence of a big accumulation of 1s in the upper tree levels, making it impossible to reduce the number of subtrees that will be visited. Therefore, S-trees are not appropriate for large page sizes.

Finally, it must be mentioned that all methods seem to converge to a point, which corresponds to the smallest query weight. This is expected since, for very small query weights, many tree nodes will match and therefore they will lead in a quite large number of page accesses, regardless of the applied method.

6.3. Evaluation of reinsertion method

Finally, we tested the use of the forced reinsertion method along with the split methods described previously. It turned out that the reinsertion method is not appropriate to be used in combination with some split methods. Therefore, here we present results only for the original linear and the quadratic split methods.

Figure 10 presents the disk accesses with respect to the query signature weight when using only the ORIG and QUAD split in comparison to the case when using the same split methods along with the forced reinsertion. As we can see in the upper part of Figure 10, for a 30% reinsertion threshold (percentage of signatures forced to be reinserted),

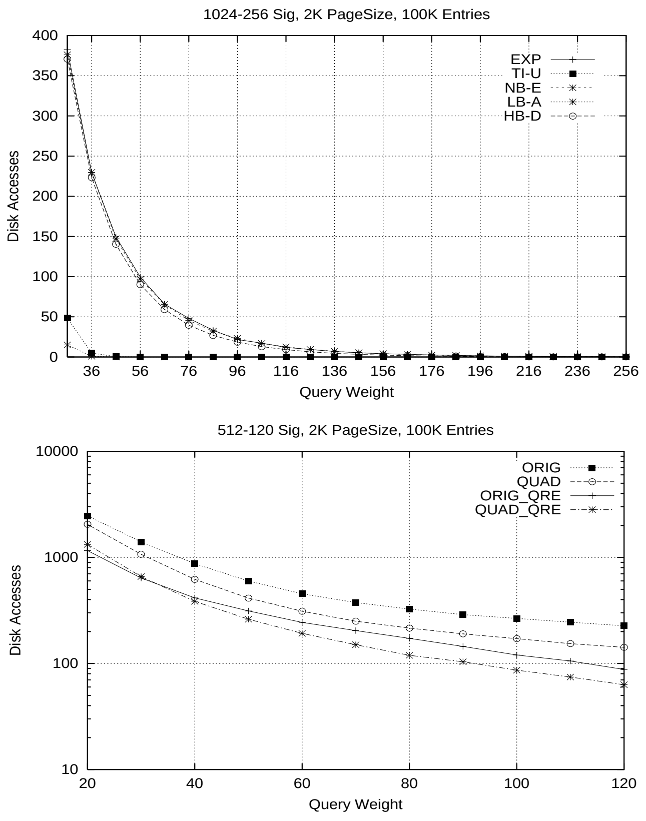


FIGURE 11. Performance of the quadratic reinsertion for larger pages (2K for 512 length signatures) and 15% reinsertion threshold.

the original reinsertion does not improve ORIG, while it worsens the performance of the QUAD one. The situation didn't improve much with a 15% threshold, neither for the original nor for the quadratic reinsertion, as shown in the lower part of Figure 10.

We also experimented with the application of the quadratic forced reinsertion in the case of double page sizes, with a 15% reinsertion threshold. The results of the original reinsertion and of the 30% reinsertion threshold did not exhibit a good performance; therefore we eliminate them from this discussion. We have already seen that when the page size is doubled, the performance of all methods deteriorates greatly. Figure 11 illustrates the disk accesses with respect to the query-signature weight for two different total numbers of signatures. We observe that when the quadratic reinsertion is applied the performance of both ORIG and QUAD split improves significantly.

The previous results, as far as the first set of experiments is concerned, were unexpected; the reorganization resulting from the reinsertion of a number of signatures, and by concurrently avoiding to perform node splitting, should improve the performance. An explanation could be that since there are only a few signatures in the small pages, the split method does succeed in clustering the signatures, whereas further reorganization from the forced reinsertion method does not pay off. On the other hand, when double page sizes are used there is still space for improvement of the clustering, and that is what the forced reorganization tries to achieve. It should be noticed, however, that due to the increased weight that appears when a large page size is used, as mentioned earlier in the section, the performance of the methods with large page sizes and reinsertion is still worse than when small page sizes are used.

7. CONCLUSION

In conclusion, we have shown that the simple split method of linear complexity is not adequate for the S-tree-access method. A number of new split methods were proposed instead. Their increased complexity is justified during query execution, as is verified by our experiments. Scale-up experiments indicated that, by applying the proposed split methods, the S-tree can be used for large signature databases.

Several new performance-estimation functions were presented. A previous function was based on the assumption of a uniform distribution of 1s at the upper tree levels. As was verified by our experiments, this assumption does not hold. We proposed an approximation of this distribution with a histogram-based method, which provides accurate results with low computational complexity in comparison to an exhaustive method.

We also examined the application of the forced reinsertion method. It seems that it does not present significant improvement and its effective use requires further consideration. Future work could involve a combination of S-trees with some of the methods that are based on data partitioning in combination with hashing techniques [36]. A possible direction could also be the application of this method in a parallel environment [37].

REFERENCES

[1] Andre-Jönsson, H. and Badal, D. (1997) Using signature files for querying time-series data. *Proc. 1st Eur. Symp. on Principles of Data Mining and Knowledge Discovery*, Trodheim, Norway, pp. 211–220.

[2] Agrawal, R., Imielinski T. and Swami, A. N. (1993) Mining association rules between sets of items in large databases. *Proc. 1993 ACM SIGMOD Conf.*, Washington, DC, pp. 207–216.

[3] Rabitti, F. and Zezula, P. (1990) A dynamic signature technique for multimedia databases. *Proc. 13th ACM SIGIR Conf.*, Brussels, Belgium, pp. 193–210.

[4] Faloutsos, C., Lee, R., Plaisant, C. and Shneiderman, B. (1990) Incorporating string search in a hypertext system: user interface and signature file design issues. *HyperMedia*, **2**, 163–174.

[5] Chidlovskii, B. and Borghoff, U. M. (1998) Signature file methods for semantic query caching. *Proc. 2nd ECDL Conf.*, Heraklion, Greece, pp. 479–498.

[6] Lee, W. C. and Lee, D. L. (1999) Signature caching techniques for information filtering in mobile environments. *ACM Wireless Networks*, **5**, 57–67.

[7] Christodoulakis, S. and Faloutsos, C. (1984) Signature files: an access method for documents and its analytical performance evaluation. *ACM Trans. Office Inform. Syst.*, **2**, 267–288.

[8] Faloutsos, C. (1992) Signature files. In Frakes, W. B. and Baeza-Yates, R. (eds), *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, Englewood Cliffs, NJ.

[9] Dervos, D., Manolopoulos, Y. and Linardis, P. (1998) Comparison of signature file models with superimposed coding. *Inform. Proc. Lett.*, **65**, 101–106.

[10] Kocberber, S., Can, F. and Paton, J. (1999) Optimization of signature file parameters with varying record lengths. *Comp. J.*, **42**, 11–23.

[11] Deppisch, U. (1986) S-tree: a dynamic balanced signature index for office retrieval. *Proc. 9th ACM SIGIR Conf.*, Pisa, Italy, pp. 77–87.

[12] Sacks-Davis, R. and Ramamohanarao, K. (1983) A two level superimposed coding scheme for partial match retrieval. *Inform. Syst.*, **8**, 273–289.

[13] Pfaltz, J., Berman, W. and Cagley, E. (1980) Partial match retrieval using indexed descriptor files. *Commun. ACM*, **23**, 522–528.

[14] Ciaccia, P. and Zezula, P. (1996) Declustering of key-based partitioned signature files. *ACM Trans. Database Syst.*, **21**, 295–338.

[15] Lee, D. L. and Leng, C. W. (1989) Partitioned signature files: design issues and performance evaluation. *ACM Trans. Office Inform. Syst.*, **7**, 158–180.

[16] Lee, D. L. and Leng, C. W. (1990) A partitioned signature file structure for multiattribute and text retrieval. *Proc. 6th IEEE Conf. on Data Engineering (ICDE'90)*, Los Angeles, CA, pp. 389–397.

[17] Zezula, P., Rabitti, F. and Tiberio, P. (1991) Dynamic partitioning of signature files. *ACM Trans. Inform. Syst.*, **9**, 336–369.

[18] Zezula, P., Ciaccia, P. and Tiberio, P. (1993) Hamming filter: a dynamic signature file organization for parallel stores. *Proc. 19th VLDB Conf.*, pp. 314–327.

[19] Ishikawa, Y., Kitagawa, H. and Ohbo, N. (1993) Evaluation of signature files as set access facilities in OODBs. *Proc. 1993 ACM SIGMOD Conf.*, Washington, DC, pp. 247–256.

[20] Kitagawa, H., Fukushima, Y., Ishikawa, Y. and Ohbo, N. (1993) Estimation of false drops in set-valued object retrieval with signature files. *Proc. 4th FODO Conf.*, Chicago, IL, pp. 146–163.

[21] Kitagawa, H., Watanabe, N. and Ishikawa, Y. (1996) Design and evaluation of signature file organization incorporating vertical and horizontal decomposition schemes. *Proc. 7th DEXA Conf.*, Zurich, Switzerland, pp. 875–888.

[22] Hellerstein, J. M. and Pfeffer, A. (1994) *The RD-Tree: an Index Structure for Sets*. Technical Report No. 1252, University of Wisconsin at Madison.

[23] Helmer, S. and Moerkotte, G. (1997) Evaluation of main memory join algorithms for joins with set comparison join predicates. *Proc. 23rd VLDB Conf.*, Athens, Greece, pp. 386–395.

[24] Lee, W. C. and Lee, D. L. (1992) Signature file methods for indexing object-oriented database systems. *Proc. 2nd Computer Science Conf.*, Hong Kong, pp. 616–622.

[25] Lee, D. L. and Lee, W. C. (1996) Signature path dictionary for nested object query processing. *Proc. IEEE Phoenix Conf. on Computers and Communications (IPCCC'96)*, Phoenix, pp. 275–281.

[26] Yong, H. S., Lee, S. and Kim, H. J. (1994) Applying signatures for forward traversal query processing in object-oriented databases. *Proc. 10th IEEE Conf. on Data Engineering (ICDE'94)*, pp. 518–525.

[27] Tousidou, E., Bozanis, P. and Manolopoulos, Y. Efficient handling of signature files used for objects with set-valued attributes, submitted.

[28] Beckmann, N., Kriegel, H. P., Schneider, R. and Seeger, B. (1990) The R*-tree: an efficient and robust access method for points and rectangles. *Proc. 1990 ACM SIGMOD Conf.*, Atlantic City, NJ, pp. 322–331.

[29] Manolopoulos, Y., Theodoridis, Y. and Tsotras, V. (1999) *Advanced Database Indexing*. Kluwer Academic Publishers, Boston, MA.

[30] Ang, C. H. and Tan, T. C. (1997) New linear node splitting algorithm for R-trees. *Proc. 5th SSD Symp.*, Berlin, Germany, pp. 339–349.

[31] Faloutsos, C. and Lin, K.-I. (1995) FastMap: a fast algorithm for indexing, data mining and visualization of traditional and multimedia datasets. *Proc. 1995 ACM SIGMOD Conf.*, San Jose, CA, pp. 183–200.

[32] Guttman, A. (1984) R-trees: a dynamic index structure for spatial searching. *Proc. 1984 ACM SIGMOD Conf.*, Boston, MA, pp. 47–57.

[33] Ciaccia, P., Patella, M. and Zezula, P. (1997) M-tree: an efficient access method for similarity search in metric spaces. *Proc. 23rd VLDB Conf.*, pp. 426–435.

[34] Eppstein, D. (1998) Fast hierarchical clustering and other applications of dynamic closest pairs. *Proc. 9th ACM-SIAM Symp. on Discrete Algorithms*, San Francisco, CA, pp. 619–628.

[35] Im, B. M., Yoo, J. S. and Kim, M. H. (1998) *A Dynamic Signature File Declustering Method based on the Signature Difference*. Technical Report, KAIST, Korea.

[36] Bozanis, P., Makris, C. and Tsakalidis, A. (1995) Parametric weighted filter: an efficient dynamic manipulation of signature files. *Comp. J.*, **38**, 478–488.

[37] Tousidou, E., Vassilakopoulos, M. and Manolopoulos, Y. (2000) Performance evaluation of parallel S-trees. *J. Database Management*, **11**, 28–34.