PERGAMON

Information Systems 27 (2002) 93–121

www.elsevier.com/locate/infosys

Signature-based structures for objects with set-valued attributes ☆

Eleni Tousidou[a], Panayiotis Bozanis[b], Yannis Manolopoulos[c,1,2]

[a] *Department of Informatics, Aristotle University, Thessaloniki 54006, Greece*

[b] *Department of Computer and Communication Engineering, School of Engineering, University of Thessaly, Argonafton & Filellinon, 38221 Volos, Greece*

[c] *Department of Informatics, University of Cyprus, Nicosia 1678, Cyprus*

Received 28 August 2000; received in revised form 5 March 2001; accepted 30 August 2001

Abstract

Aiming at the efficient retrieval of objects with set-valued attributes, we introduce three variations of a new method in order to satisfy subset and superset queries. Our approach is to combine the advantages of two access methods, that of linear Hashing and of tree-shaped methods, on which other similar methods have been previously reported as well. Performance estimation analytical functions for each particular method are presented, followed by a thorough experimental comparison of all investigated structures, where analytical and experimental results deviate 10% on the average. Finally, the results of this performance evaluation are presented and discussed, clearly showing the superiority of the new methods reaching an improvement of up to 85%. © 2002 Published by Elsevier Science Ltd.

1. Introduction

Advanced database application areas, such as computer aided design, office automation and multimedia data present the need to manipulate efficiently complex objects. Object-oriented databases can come to the aid of such areas that contain both formatted and unformatted, complex data by supplying set and tuple constructors and by making use of the existing index structures. The basic query for this kind of data is the *inclusive* (or *partial match*) query, which retrieves all records containing specific attributes. Inclusive queries can be divided in subset queries, i.e. queries that search for all stored sets that contain the query set, and in superset queries, i.e. queries that search for all stored sets that are contained in the query set. The focus of this paper is in subset queries, whereas the case of superset queries will be also discussed. An example of a subset query would be "*Find all car-owners who have in their ownership at least a Mercedes and BMW*", whereas on the other hand, an example of a superset query would be "*Find all car-owners who have in their ownership no other car except of a Mercedes or a BMW*".

☆ Recommended by Yannis Ioannidis, Area Editor.

E-mail addresses: eleni@delab.csd.auth.gr (E. Tousidou), pbozanis@inf.uth.gr (P. Bozanis), manolopo@ucy.ac.cy (Y. Manolopoulos).

[1] On sabbatical leave from the Department of Informatics, Aristotle University, Thessaloniki, Greece 54006.

[2] Present address: Department of Informatics, University of Cyprus, Nicosia 1678, Cyprus.

0306-4379/02/$ - see front matter © 2002 Published by Elsevier Science Ltd.
PII: S0306-4379(01)00047-3

During the last few years, several indices have been proposed in order to support the manipulation of object-oriented and object-relational data models. In [1–5] the presented structures manage to cope with path expressions, which are fundamental in object-oriented databases, but none of them has dealt with data that contain set-valued attributes. Consequently, these structures cannot readily deal with the above kind of queries. Due to this need, the use of signature files was introduced. Set-valued attributes are now being represented by bit vectors, called *signatures*. Signature files are collections of signatures and have been widely accepted due to their low space overhead and reduced update costs [6,7]. However, since signatures are abstractions of the original represented information, they introduce an information loss. As a result, non-qualifying objects, called *false drops*, may be retrieved as well, which are discarded though after a filtering stage.

Works on indexing signature files can be generally divided into two categories: the exhaustive approach and a more restrained one. In order to achieve a good performance in the first case, researchers had focused in developing good signature extraction methods for the reduction of false-drop probability and the improvement of the retrieval performance in signature files [6,8,9]. A more efficient approach based on the prevention of reading unnecessary signature bits was introduced in [10–12], where several variations of the bit-sliced signature file are presented.

In the second case, trying to totally avoid sequential searching, a partitioning of the superimposed signature files was proposed. Horizontal or vertical fragmentation in combination with hashing techniques were studied in [13–15]. A similar approach has been investigated by Zezula et al. who proposed the method of quick filter [16]. In [17], the hybrid structure of parametric weighted filter (PWF) was introduced taking advantage of the hash-based partitioning methods. In this work, as in [16], hash buckets contain not only a single partition page but a specified number of pages with the aid of superimposition ("or-ing").

A different direction was the use of tree-based signature file organizations such as the two-level [18] and multi-level [19] index, and the most popular one, the S-tree [20]. S-trees are height balanced dynamic structures similar to B^+-trees that have been proposed to improve the searching performance in signature-based organizations. In [21] an improvement of the S-tree construction is presented in order to achieve a better clustering of the stored signatures. RD-trees have also 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], the use of signatures in path expressions has been studied as well. Recently, signatures have also been used in order to perform join operations [26] between two relations containing set-valued attributes.

Common ground in most of the previous methods is that they were originally presented for use in text databases. Only lately do authors refer to the particular properties, needs and problems that are met in object-oriented databases, such as the constrained domain of set-valued attributes in contrast to the non-constrained one of text-databases, and the superset, perfect match and subset queries that are frequently asked. In this paper, we focus on the advantages that could arise from the creation of a hybrid structure based on the advantages of hashing and the S-tree as well. In [16,17], the authors have introduced a similar hybrid structure, where the only clustering involved had as a measure of similarity a suffix of usually limited length, which played the role of a hash table key. Here, we try to increase the similarity of grouped signatures even more by using the similarity functions that are introduced in [20,21].

The rest of the paper is organized as follows. Section 2 gives a brief description of signature creation, and of the two competing methods S-tree and the PWF. Section 3 contains the observations that led us to the introduction of the new methods. These methods are described in detail in Section 4. Two approaches of estimating each method's behavior are described in Section 5, while in Section 6 a thorough comparison of both theoretical and experimental evaluation is presented. Finally, Section 7 contains some concluding remarks and directions for future work.

2. Background

2.1. Signatures

Signature, symbolized by s, is a bitstring of F bits, where F is the signature *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, which will set exactly m out of F bits to 1, where the value of m is called *weight* of the element signature and is often symbolized by $\gamma(s)$. The set bits are uniformly distributed in the $[1..F]$ 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 one 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 an 1 in all positions, where the query signature has also an 1.

A superset query searches for all objects containing at least some of the queried attributes and only those. Here, the answer to this query is the collection of all objects o for which $o' \supseteq o$. It is similarly easy to show that

$$o' \supseteq o \Rightarrow q \supseteq s,$$

where the right part of the above expression represents the fact that there is no position in the signature s having a value of 1 when the query signature has a 0 in the corresponding position.

Thus, signatures can provide a filter for testing subset and superset queries on attributes, because if the subset (superset) condition does not hold for the signature, then it does not hold for the object neither. 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 [6]:

$$F \times \ln 2 = m \times D, \tag{1}$$

where D is the number of object attributes in a stored set. When this condition is applied, the signatures that will be created will contain 50% 1s and 50% 0s, on the average.

For example, suppose we want to store a set Σ comprised of sets of car brands that certain people own, in order to be able to answer efficiently subset and superset queries on these sets. Using an appropriate hash function, we encode each brand name as a bit-string of length 16; the resulting signatures are as follows:

Land Rover	$\mapsto$	0000 0000 0010 0001
BMW	$\mapsto$	0000 0000 0100 0001
Lancia	$\mapsto$	1000 0000 0100 0000
Toyota	$\mapsto$	0001 1000 0000 0000
Nissan	$\mapsto$	0010 0100 0100 0000
Chrysler	$\mapsto$	0001 0001 0000 0000
Ferrari	$\mapsto$	1100 0000 0000 0000

Renault	↦	0000 0000 1100 0100
Cadillac	↦	0000 0011 0000 0000
Jeep	↦	0000 0100 0011 0000
Opel	↦	0000 0000 1000 0001
Mercedes	↦	0000 0100 0100 0000
Alfa Romeo	↦	0000 1000 0000 0001
Hyundai	↦	0000 0000 1000 1001
Citroën	↦	0001 0010 0000 0000
Pontiac	↦	0000 0000 0000 1001
Daewoo	↦	0010 0000 1000 0000
Seat	↦	0010 0010 0000 0000
Peugeot	↦	0000 0010 0011 0000
Volvo	↦	0000 0000 0000 0100

Based on these brands, the sets of cars contained in Σ will be mapped to signatures as follows:

{BMW}	↦	0000 0000 0100 0001
{Mercedes}	↦	0000 0100 0100 0000
{Seat}	↦	0010 0010 0000 0000
{Daewoo}	↦	0010 0000 1000 0000
{Opel}	↦	0000 0000 1000 0001
{Renault}	↦	0000 0000 1100 0100
{Toyota, Hyundai}	↦	0001 1000 1000 1001
{Nissan, BMW, Pontiac}	↦	0001 0010 0100 1001
{BMW, Nissan, Citroën}	↦	0011 0110 0100 0001
{BMW, Mercedes, Opel}	↦	0000 0100 1100 0001
{Lancia, Ferrari, BMW}	↦	1100 0000 0100 0001
{Lancia, Peugeot, BMW}	↦	1000 0010 0111 0001
{Ferrari, BMW}	↦	1100 0000 0100 0001
{BMW, Mercedes}	↦	0000 0100 0100 0001
{Cadillac, BMW}	↦	0000 0011 0100 1001
{Opel, Volvo}	↦	0000 0000 1000 0101
{Jeep, Land Rover}	↦	0000 0100 0011 0001
{Chrysler, Pontiac}	↦	0001 0001 0000 1001
{Lancia, Alfa Romeo}	↦	1000 1000 0100 0001
{Daewoo, Volvo, Renault, BMW}	↦	0010 0000 1100 0101

In this way, the queries

$$q_1 = \{\sigma \in \Sigma \mid \sigma \supseteq \{\text{Mercedes, BMW}\}\} \quad \text{and} \quad q_2 = \{\sigma \in \Sigma \mid \sigma \subseteq \{\text{Mercedes, BMW}\}\}$$

are transformed to subset and superset queries, where $q = 0000\ 0100\ 0100\ 0001$ is the query's bitstring. The sets {BMW, Mercedes}, {BMW, Nissan, Citroën} and {BMW, Mercedes, Opel} are returned as an answer to query q_1 (the second one being a false drop) and the sets {Mercedes}, {BMW} and

{Mercedes, BMW} are returned to q_2. The reader can see in Fig. 1 how Σ is actually accommodated in an S-tree.

Finally, we say that signatures a and b are more *similar* than signatures a and c, when a and b's superimposed signature has less weight than the one produced by a and c, or else when:

$$\gamma(a \vee b) < \gamma(a \vee c).$$

2.2. S-tree

S-trees, similarly to B^+-trees, are height balanced trees having all leaves at the same level [20]. 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 2 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: $1 \leqslant k \leqslant K/2$. The height of the tree 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 object signatures, the S-tree may contain duplicate signatures corresponding to different objects. In Fig. 1, an example of an S-tree with height $h = 3$ is depicted. The signatures appearing in the leaves represent individual set signatures, i.e. the indexed objects, of set Σ, described previously.

Successful searches in an S-tree proceed as follows. Given a user query for an object, we compute its signature and compare it to the signatures stored in the root. For example, in case of a subset query, 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

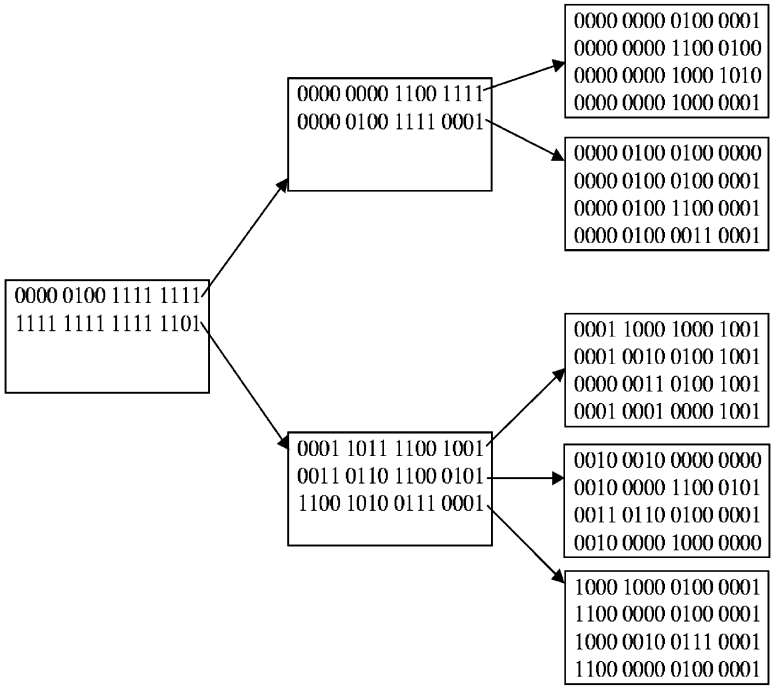

Fig. 1. An example of an S-tree with $K = 4$ and $k = 2$.

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 which may be the desired ones (after discarding false-drops). In case of an unsuccessful search, searching may stop early at some level above the leaf level, if the signature of the user query has 1s at positions where the stored signatures have 0s.

Due to the superimposition technique, nodes near the root tend to contain heavy signatures (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 [20], 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-tree is affected by the disk accesses to the leaf nodes.

2.2.1. Insertion and clustering techniques used in S-tree

As described in [20], the leaf that will accommodate a new signature is selected by traversing the tree top-down and by 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 [20]

$$\varepsilon(s, s') = \gamma(s \vee s') - \gamma(s). \tag{2}$$

Thus, selecting the node with minimum ε aims at the minimization of the number of multiple paths that have to be followed during searches. As a tie criterion, the node with minimum hamming distance may be used. After an insertion in a leaf follows the respective update at upper levels.

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, or else, so that the more similar signatures are grouped together.

The splitting algorithm described in [20] 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 the signature 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, 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.

In order to reduce the low selectivity problem mentioned earlier in the section, a number of improved split methods of quadratic and cubic complexity are introduced in [21], which perform up to 5–10 times better when used in partial match queries. In the present paper we adopt one of these methods, namely the *Quadratic Split*, since it manages to balance between an efficient behavior and a quadratic complexity, as its name states. In particular, we retain the original algorithm for choosing seeds but we change the way we distribute the remaining signatures to the two nodes. 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. Although it presents an increase in time complexity, $O(n^2)$ compared to the $O(n)$ overhead of the original linear algorithm, where n are the node entries, it performs a more careful signature assignment to nodes compared to the linear method, resulting into decreased node weights. Testing this algorithm showed

that splitting nodes by this method performed on the average about 3 times better than when they were split by the linear one [21] when used in subset/superset queries.

2.3. Parametric weighted filter

Before describing the way the PWF works, a brief overview should be given of the quick filter technique [16] on which it is based. Quick filter considers key signatures as binary numbers and takes their suffixes to determine the address of a linear Hashing file [27] into which the signatures are stored. For example, suppose that at a specific point in time, the file size is N pages. In such a case, there are 2^h–N pages addressed with a signature key suffix of h–1 bits, whereas there are $2N$–2^h pages which are addressed with a key suffix of h bits. The parameter h is called *hash level* and satisfies the equation $2^{h-1} < N \leqslant 2^h$.

The PWF, also, uses the method of linear Hashing in order to partition the signature file. However, the basic innovation is that each partition, i.e. each hash table address, corresponds not only to one, but to a specific number of pages. This number is controlled in order to attain the desired performance, while the pages are preallocated upon creation of a new entry in the hash table.

For example, in Fig. 2 a PWF structure is depicted, where the length of the stored signatures is 16 and the hash level is 2. The disk blocks at the right part of the figure are called *partition pages* and have a capacity of nine signatures each. The signatures of a partition page are grouped according to their arrival time into logical pages of b signatures ($b = 3$ in our example). Such a group is named *b_group* and is characterized by a *representative*, which consists of two parts: the *s_descriptor* and the *m_descriptor*. The s_descriptor is

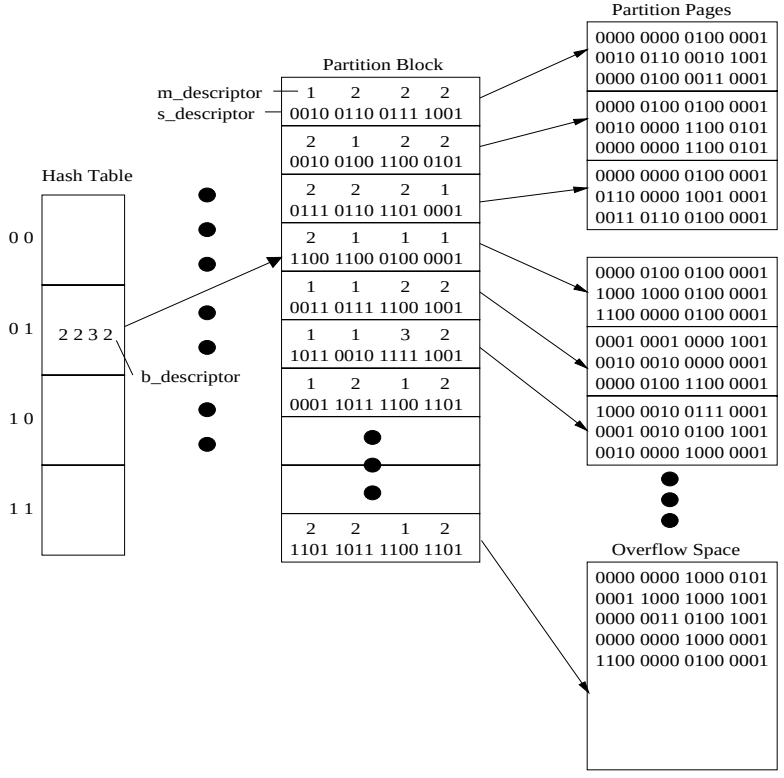

Fig. 2. An example of PWF with $F = 16$ and $h = 2$.

formed by superimposing ('or-ing') the signatures of the b_group, whereas the m_descriptor is a $\lfloor \log F \rfloor$-tuple obtained by using the following procedure. Each signature of the b_group is divided into $\lfloor \log F \rfloor$ parts each $\lceil F/\lfloor \log F \rfloor \rceil$ bits long, except for the last one which is $F - \lceil F/\lfloor \log F \rfloor \rceil(\lfloor \log F \rfloor - 1)$ bits long. Then the ith component of the m_descriptor is the maximum number of 1's (maximum weight), a signature in the b_group has in its ith part. Thus, the m_descriptor of the first b_group of the first partition page takes the form $(1, 2, 2, 2)$, since 1 is the maximum number of 1s in the first $\lfloor \log F \rfloor$ part of the signatures of this b_group, while the rest are 2 for the same reason.

All representatives are stored at a higher level structure as depicted in the middle part of the figure. Actually, representatives of the same partition page are stored into the same *partition block*, which corresponds to a disk block and contains representatives of the same partition. In an analogous manner, a descriptor, named *b_descriptor*, is extracted similarly to the m_descriptor for each partition block. This b_descriptor is stored in the hash table, as depicted in the left part of the figure.

Searching for a signature in PWF is done in two steps. In the first step, we locate the appropriate hash table position according to the signature suffix and check its b_descriptor whether it will refrain us from actually accessing the corresponding partition block. If it does not, then at a second step we visit the partition block and, thus, we retrieve only the partition pages having qualifying representatives. For example, in the same figure, in case the signature 0010 0010 0110 0101 was queried, it would pass through the '01' entry of the hash table according to its suffix, since the query b_descriptor $(1, 1, 2, 2)$ is 'smaller' (or 'equal') than the relevant entry $(2, 2, 3, 2)$. In the sequel, it would not pass through the first entry of the partition block, since its s_descriptor 0010 0110 0111 1001 does not match the query signature. Also, it would not pass through the third entry of the partition block, since its m_descriptor $(2, 2, 2, 1)$ does not match the m_descriptor of the query signature either.

Insertion of a new signature is performed by first selecting the appropriate partition block from the hash table, and then the partition page with the least number of stored signatures. After the new signature is stored in the partition page, the corresponding representative has to be updated to reflect the latter insertion. In case of a full partition block, the representative is inserted into the overflow area of that partition and the hash table is expanded by one entry. PWF structure has a global overflow space for each partition block, with theoretically unlimited capacity, which is represented by a single representative as shown in Fig. 2. This space has to be searched sequentially if its representative allows the query to proceed to the second step. However, it should be noted that this is a theoretical scenario since, in practice, overflow space is maintained during partition page splits. When the hash table is expanded, the signatures that belong to the partition that will be split, are redistributed between the two new partitions according to their new suffix. Therefore, b_groups and the overflow space will be reorganized and the new representatives will be inserted in the two new partition blocks.

3. Performance factors and motivation

From the previous section it becomes apparent that the PWF method can be extended quite efficiently if attention is paid to some of its properties. First of all, signature clustering in the partition pages can be improved with respect to the involved similarity according to the definition given in Section 2.1. Although PWF is a packed method, the only care of similarity taken is during the hash table traversing, by using the suffix as a key. The whole value of a signature instance is used only during searching when we examine the m_descriptor. Taking advantage of the signature as a whole upon insertion as well, would improve the similarity and, therefore, the searching performance, since selectivity would raise and the overlap between nodes would decrease.

A second point in PWF is the fact that the overflow space is searched sequentially. Organizing it in distinct pages would eliminate sequential scanning and thus would improve performance. A similar effect

could be achieved if pages were not preallocated for each partition block, but if instead they were given upon demand triggered during splits.

A final point is the fact that PWF does not handle superset queries, no reference having been done about this query in [17]. Towards this end, *l_descriptors* will be employed; a l_descriptor stores in its *i*th component the minimum number of 1's (minimum weight), a signature in the b_group has in its *i*th part. In [17] they were suggested as an auxiliary mechanism during the insertion procedure, if one could accept the storage overhead, but no experimental evidence was given. In Section 6 it will be shown how l_descriptors could be used during superset queries. Actually they are proven as the only means — apart from the hash key — that help avoiding the traversal of branches of the index in the case of a superset query. Also, in the case of S-trees, it is true that the whole structure has to be scanned when trying to answer such a query [20]. Recall also from S-trees that removing the upper tree levels (and actually maintaining only the lower levels) would avoid accessing nodes with very heavy signatures and would provide increased performance.

Therefore, in the present paper we focus on a better organization at the leaf level and a better usage of overflow space, in order to combine particular advantages of each method and improve the retrieval performance. Along these lines, in the next section we present three new structures. In the first method, named *Linear Hash partitioning with S-tree split* (to be referred LHS in the sequel), signature insertions and page handling follow the methods used in S-trees. After experimenting with this structure and also having in mind the results of [21], it was evident that superimposing a big number of signatures obscures selectivity and, consequently, decreases performance. Therefore, employing logical pages of small size seemed a compelling idea. When referring to logical pages we mean the creation of groups of entries inside a disk (physical) page, which will act as independent pages up to a certain degree. For example, the way b_groups exist in the PWF partition pages is actually a kind of logical pages. However, the creation of such pages in an S-tree like structure is not a straightforward issue. A first attempt was made by the second method named *Linear Hash partitioning with S-tree split and Local reorganization* (LOC in the sequel), where each page contains up to two, partly independent, as will be shown later, logical pages. This method, achieves a quite better performance than LHS, but is constrained to use only two logical pages per physical page. This problem was coped with by the third structure, the *Linear Hash partitioning with S-tree split and Logical pages* (LOG in the sequel), where the use of more independent logical pages results in finer signature clustering, at the cost though of a partly increased storage overhead.

4. Proposed methods

4.1. Linear Hash partitioning with S-tree split

After noticing that PWF inserts new signatures according to their arrival time, meaning that it groups signatures without paying attention to their similarity, an effort to improve this clustering could result in a better search performance. Such a promising attempt would be to apply the rules, which are used for the clustering of signatures in S-tree leaves.

More specifically, in a straightforward way, each entire partition page would be represented by a single couple of s_ and m_descriptors stored in the partition block, as shown in Fig. 3. This is straightforward because pages are not divided into b_groups but each entire page is actually an autonomous b_group. Additionally, although the partitioning of the signature file, as created by hashing, will remain the same, signatures will not be randomly inserted into a page. Actually, after the appropriate partition block is located from the hash table, the selection of the partition page will be based on the increase of the respective s_descriptor's weight. That is, we try to minimize the same function ε, as the one that is used in the

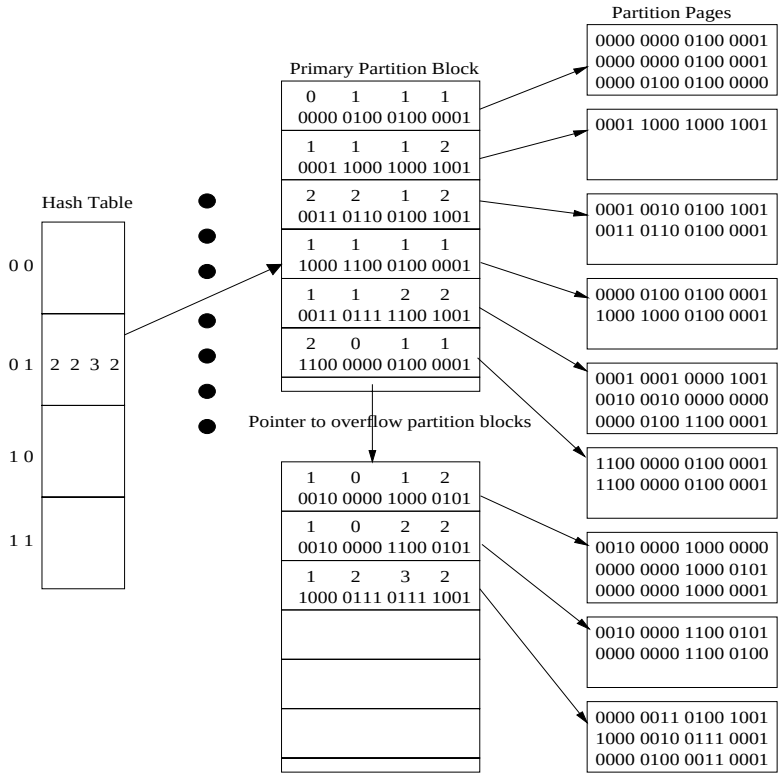

Fig. 3. An example of LHS structure with $F = 16$ and $h = 2$.

S-tree (see Eq. (2)):

$$\varepsilon = \gamma(s \vee s') - \gamma(s),$$

where s' is the new signature and s is the s_descriptor of a leaf node.

After accessing the appropriate partition page, if there is enough free space, the signature will be inserted and the corresponding representative will be updated; otherwise, the page will be split. At this point, we apply the quadratic split method, as described in Section 2.2. The resulting two new pages will then produce two new representatives and the partition block will be updated.

Overflows of partition blocks are handled differently than what applies in the PWF method. Namely, each *primary* partition block, i.e. the first block that is created upon creation of a new hash entry, will have a pointer pointing to the first overflow block in a list of overflow blocks (see Fig. 3). Entries in these overflow blocks will be representatives of partition pages that belong to the same partition but could not be accommodated in the primary partition block. The property that each partition block contains representatives of the same partition pages, and that each partition page belongs to one partition, still holds.

During the expansion of the address space, no internal reorganization is performed since the whole page can now be attached to the appropriate partition block according to its representative. Here, it must be mentioned that there could be a case where a signature is not placed to the appropriate partition block but to its sibling, as a side effect of superimposition. For example, in Fig. 3, the last signature of the first page has a 00 suffix and not a 01 as suggested by the partition. This though is a legal case, as is more clearly

explained in the appendix. In the sequel, we describe shortly the steps that have to be followed during an insertion.

Insertion algorithm in LHS:

1. Create the hash key according to the quick filter.
2. Retrieve the respective primary partition block.
3. Go through all entries in the block (and its overflow pages) checking for the most appropriate entry (minimize ε).
4. Retrieve the respective partition page.
5. If there is still space in the page, then insert the new signature and update the representative. Else, split the page according to the quadratic split method and post the new representative into the primary partition block.
6. If there is free space in the primary partition block, then insert the new representative. Else,
   insert it in the partition's overflow space;
   reach the partition that is next to be split according to linear hashing;
   redistribute the entries of the primary block and its overflow pages into the two new partitions according to their new suffix.

Searching is performed in exactly the same way as in the PWF method. This means that, firstly, the hash table will lead to the appropriate partition blocks and, from there on, qualifying representatives, with respect to the query signature, will trigger the access of the partition pages that contain candidate objects.

In the following two methods, overflows and searching are handled in exactly the same way as described here. What is also common in all three new methods is that partition pages are only allocated upon demand during the insertion process and not upon the creation of a new partition block as in the case of PWF. What changes is the way that b_groups, i.e. logical pages, are created and placed in each physical page.

4.2. Linear Hash partitioning with S-tree split and local reorganization

Experimenting with the previous method and having in mind the results from [21], we wanted to test the behavior of a similar structure where each representative would be produced by a smaller number of signatures, considering that this would decrease its weight and would increase selectivity. Our first approach was based on the use of logical pages, similar to the b_groups of PWF method, that would be stored in a physical page and would be locally reorganized.

More specifically, now new signatures are being inserted in a specific node until half of the node is filled. This means that if the node has capacity P, then upon insertion of the $(P/2+1)$th entry a virtual split will take place as if the node had overflown, where the $(P/2+1)$ entries will be equally divided between the two new nodes using the quadratic split method. We use the word 'virtual' because no new physical page is created but only logical pages are reorganized. For example, if a is the original node and b, c are the two resulting nodes, then the entries of b will be inserted in the first half of node a, while the entries of c will be inserted in the second half. Consequently, each node will now be equally divided into two parts, creating this way two b_groups, or else, two logical pages. Each logical page produces a representative, which will be stored in the respective partition block.

From there on, insertions can take place either in the first or in the second logical page according to their representatives. Thus, after an insertion, the respective representative is updated. Logical pages do not behave independently but, each time one of them overflows (contains more than $P/2$ entries), the node is similarly virtually split and the signatures are internally rearranged into the two existing logical pages,

followed again by the update of the respective representatives. When an overflow of the whole node takes place, it will be actually split this time and the two new leaves that will be created, will be similarly divided into logical pages. In the following, the steps of the insertion algorithm are described.

Insertion algorithm in LOC:

1. Create the hash key according to the quick filter.
2. Retrieve the respective primary partition block.
3. Go through all entries in the block (and its overflow pages) checking for the most appropriate entry (minimize ε).
4. Retrieve the respective partition page and, therefore, the respective logical page.
5. If there is still space in the logical page, then insert the new signature and update the representative.
   Else, if the logical page is full (contains $P/2$ entries) while the respective node is not, virtually split in half the whole node according to the quadratic split, and accommodate the two new logical pages in the first and second half of the node.
   Else, split the node, and similarly create logical pages in the two resulting nodes.
   Update the representatives and post the new ones in the primary partition block.
6. If there is free space in the primary partition block, then insert the new representatives.
   Else,
   insert it in the partition's overflow space;
   reach the partition that is next to be split according to linear hashing;
   redistribute the entries of the primary block and its overflow pages into the two new partitions according to their new suffix;
   if a page produces representatives that belong to different partitions, then split the page in half and update the corresponding blocks.

In Fig. 4, such a structure is shown where each physical page has a capacity of 6 entries, creating therefore two logical pages of maximum three entries. Focusing at the second physical page, if a signature was about to be inserted in its second logical page, there would be a local overflow and the signatures would be rearranged internally into two new logical pages of 3 entries each, following the split procedure used in S-tree. If yet another signature was about to be inserted in this page, then the node should split and two new partition pages be created since there is no free space.

On the other hand, when a partition block is split, its stored representatives are rearranged into two new partition blocks according to the new suffix. In case that a page, which originally belonged to the split partition block, produces representatives that will now belong to different partitions, it will have to be split so that each new page will belong only to one partition block and each block will contain representatives of the same partition.

4.3. Linear Hash partitioning with S-tree split and logical pages

The previous method succeeds in lowering the number of signatures from which a representative is extracted and therefore it increases selectivity. However, it confines the number of logical pages that can be stored in each partition page to two and also restrains their independence since they are reorganized whenever anyone of them overflows. By relaxing these constraints, a third method was introduced, namely *Linear Hash partitioning with S-tree split and logical pages*, where a number of independent logical pages are stored in a partition page, with the latter corresponding to a physical page. As it will be shown, this number is a parameter that can be tuned in order to achieve the required performance and trade off between space overhead and retrieval cost.

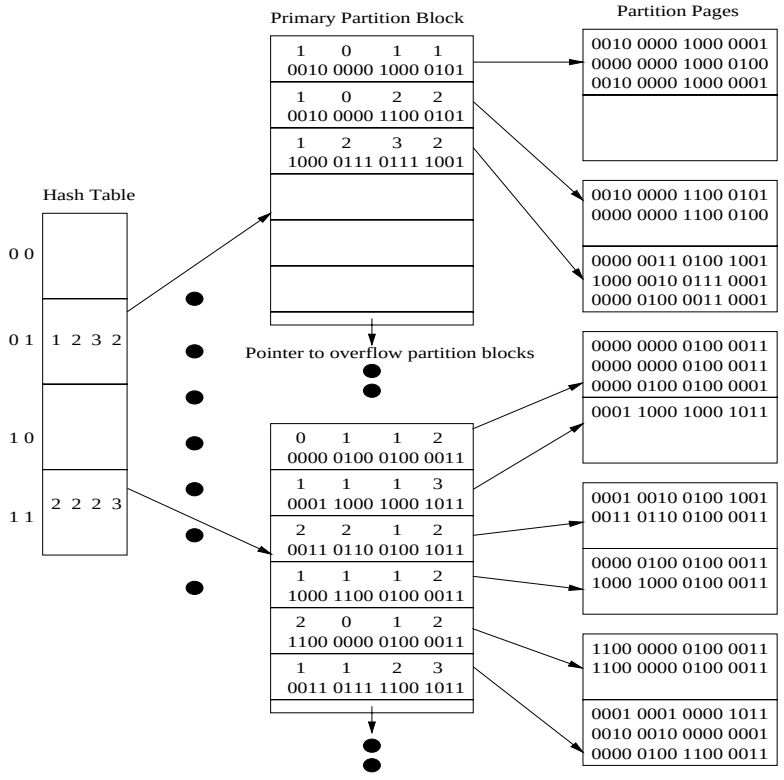

Fig. 4. An example of LOC structure with $F = 16$ and $h = 2$.

Now, each partition block will lead to a number of physical pages in which a specified number of logical pages will exist. As shown in Fig. 5, logical pages will produce their representatives which will be stored in possibly different partition blocks. Thus, logical pages will be treated as independent pages since they will only be changed upon insertion through their representatives in the partition blocks and will not interact with each other.

More specifically, signatures keep being inserted in an empty physical page, until the first logical page is filled. If yet another signature is to be inserted in the same logical page, an overflow will occur. Here, no internal reorganization will take place but instead, the logical page will be split by following the quadratic split method and one more logical page will be created. This new logical page will be stored in the same physical page. However, if there is lack of space, then it will be stored in *any* physical page that can accommodate it; otherwise, a new physical page will be created. As a result, some kind of packing is performed as far as the relation between logical pages and physical pages is concerned. The respective steps of the insertion algorithm are shown in the following.

Insertion algorithm in LOG:

1. Create the hash key according to the quick filter.
2. Retrieve the respective primary partition block.
3. Go through all entries in the block (and its overflow pages) checking for the most appropriate entry (minimize ε).

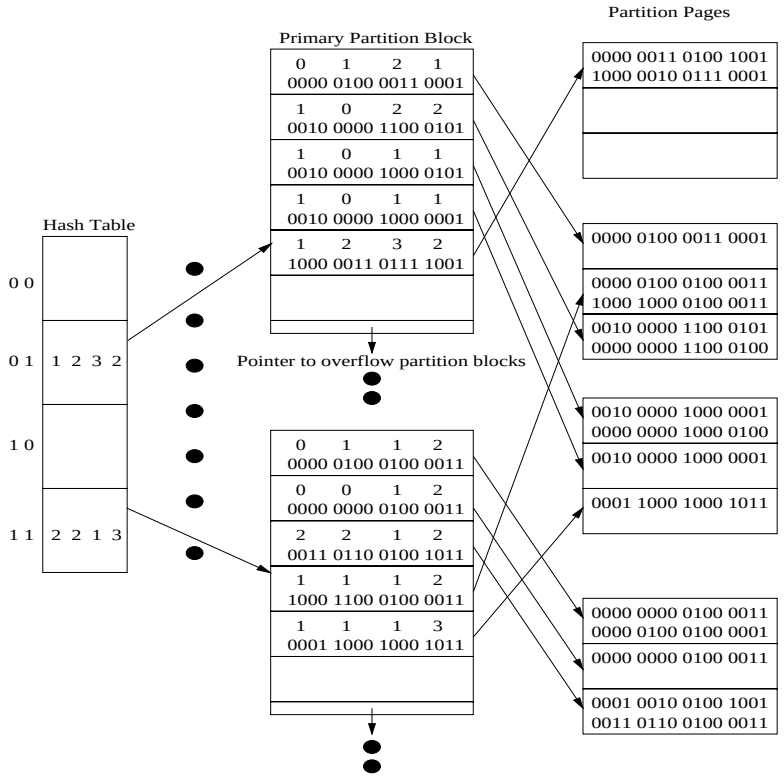

Fig. 5. An example of LOG structure with $F = 16$ and $h = 2$.

4. Retrieve the respective partition page and, therefore, the respective logical page.
5. If there is still space in the logical page, then insert the new signature and update the representative.
   Else, split the logical page and enter the new one in a physical page that can accommodate it. If there is not any, then get a new physical page.
   Update representatives and post the new one in the primary partition block.
6. If there is free space in the primary partition block, then insert the new representative.
   Else,
   insert it in the partition's overflow space;
   reach the partition that is next to be split according to linear hashing;
   redistribute the entries of the primary block and its overflow pages into the two resulting partitions according to their new suffix.

For example, in Fig. 5, each physical page can accommodate three logical pages of maximum two entries. Each logical page creates its own representative, which is stored in the corresponding partition block. Looking at the last partition page, if a signature is to be stored in the first or third logical page, any of the latter ones would overflow. Thus a split should occur and a new logical page should be created. Since this partition page cannot accommodate the new page, it will be stored in the first one which still has free space.

Two issues should be noticed in this approach. The first is that an S-tree like split takes place only upon a logical page overflow, whereas there is no split of partition pages. New partition pages are only created upon demand triggered by the lack of space for a logical page. The second observation, which comes as a consequence of the way partition blocks are split, is that physical pages play the role of the storage place of the logical ones. Upon a split of a partition block, its entries are appropriately redistributed to the two new partition blocks while their references to partition pages remain the same. This means that there might be a case where entries in different partition blocks will refer to the same partition page. Therefore, although there is an one-to-one correspondence between representatives, i.e. logical pages, and partition blocks, there can be no such connecting criterion between partition blocks and partition pages.

5. Performance estimation

Assuming that signatures are extracted using the superimposition technique as described in [6,10], we are interested in estimating the number of page accesses required to satisfy a subset or a superset query. Perfect match queries are an instance of subset queries. In the following, we estimate this cost for subset queries whereas the performance of the remaining queries can be similarly estimated.

Two different methodologies have been followed in order to derive the cost formula. The first one is based on the approach of [17]. The cost is considered as a combination of accessing the partition blocks of linear hashing and of accessing the group of leaves that belong to each partition block, constrained though by the existence of the m_descriptor. The second approach is the one described in [20], where page accesses are counted based on the probability that the queried signature will match with stored signatures while descending the tree from the root to the leaf level.

At this point, it should be mentioned that for the rest of the analysis we will refer to the level of the hash table as level 1, to the level of partition blocks as level 2 and to the level of partition pages as level 3. All symbols that will be used in the following are summarized in Table 1.

According to the first approach, when traversing the structure from the third level to the first one, each of the proposed methods will contain $\bar{n}_3$ signatures per partition page on the average, where:

$$\bar{n}_3 = \frac{P}{F+p}\alpha_3. \tag{3}$$

This product is the result of not having fully packed pages but instead presenting an α_3 load factor at the leaf level. However, the number of representatives that are produced from each page will differ according to the method under examination. For the first method, LHS, each page produces only one representative, whereas for the second one, LOC, on the average one and a half representatives are produced since a page can contain up to two b_groups. In the third method, LOG, the number of representatives depends on the parameter b. Therefore:

$$\bar{n}_{rp} = \begin{cases} 1 & \text{for the LHS method,} \\ 11.5 & \text{for the LOC method,} \\ \bar{n}_3/b & \text{for the LOG method.} \end{cases} \tag{4}$$

Ascending one level up, the partition blocks will store representatives and pointers. According to Section 2.4, the length of each m_descriptor is

$$len(m\_desc) = (\lfloor \log F \rfloor - 1)\left\lceil \log\left(\left\lceil \frac{F}{\lfloor \log F \rfloor}\right\rceil\right)\right\rceil + \left\lceil \log\left(F - \left\lceil \frac{F}{\lfloor \log F \rfloor}\right\rceil(\lfloor \log F \rfloor - 1)\right)\right\rceil$$

Table 1
Symbol table

Symbol	Definition
F	signature length
n	total number of signatures
P	page size in bits
p	pointer size in bits
l	hash level of linear hashing
b	grouping factor in partition pages
PT	total number of partition blocks
α_j	load factor of nodes at level j, for $1 \leqslant j \leqslant 3$
γ_i	weight of the superimposed signature of a logical page i
γ_q	query signature weight
$\gamma_q(l)$	query signature weight of an l-bit substring
p_1, p_0	probability of a signature bit to be set or not, respectively
n_2	number of representatives per partition block
n_3	number of signatures per partition page
n_4	number of signatures per b_group
n_{rp}	number of representatives per partition page
n_{pb}	number of pages represented by a partition block
n_{sb}	number of signatures represented by a partition block
$\bar{x}$	average value of quantity x

resulting in storing:

$$\bar{n}_2 = \frac{P}{F + p + len(m\_desc)}\,\alpha_2 \tag{5}$$

representatives per partition block. Consequently, having in mind that the average number of pages that are accommodated in a partition block is [17]

$$\bar{n}_{pb} = \frac{\bar{n}_2}{\bar{n}_{rp}} \tag{6}$$

and assuming that F is reasonably long (i.e. $F > 100$), the number of signatures that are actually represented in a block is

$$\bar{n}_{sb} = \bar{n}_{pb} \cdot \bar{n}_3 = \begin{cases} \bar{n}_2 \cdot \bar{n}_3 & \text{for the LHS method,} \\ \frac{2}{3} \cdot \bar{n}_2 \cdot \bar{n}_3 & \text{for the LOC method,} \\ b \cdot \bar{n}_2 & \text{for the LOG method.} \end{cases} \tag{7}$$

Reaching the first level, the number of blocks that need to be retrieved by the quick filter on the average is [16]

$$hashAcc = 2^{l - \bar{\gamma}_q(l)}, \tag{8}$$

where l is the hash level and 2^l is the number of data pages. However, due to the use of the m_descriptor, not all qualified blocks will finally be accessed. The activation of a specific group containing t signatures due to their m_descriptor is [17]

$$p_{act}(t) = \left\{ 1 - P\left(\mathbf{x} < \frac{\bar{\gamma}_q}{\lfloor \log F \rfloor} \right)^t \right\}^{\lfloor \log F \rfloor},$$

where

$$P\left(\mathbf{x}<\frac{\bar{\gamma}_q}{\lfloor \log F \rfloor}\right)=\sum_{i=0}^{\frac{\bar{\gamma}_q}{\lfloor \log F \rfloor}-1}\binom{\frac{F}{\lfloor \log F \rfloor}}{i}p_1^i\, p_0^{\frac{F}{\lfloor \log F \rfloor}-i}, \tag{9}$$

where p_1 and p_0 are the probabilities for a bit position to be set to one and zero, respectively, and $\mathbf{x}$ is a random variable representing the number of 1s in a sub-signature of length $\lceil F/\lfloor \log F \rfloor \rceil$. Variable $\mathbf{x}$ follows the $\mathscr{B}(F/\lfloor \log F \rfloor, p_1)$ binomial distribution.

At the same level, apart from the m_descriptor, the probability for an s_descriptor with weight $\overline{\gamma_i}$ to be activated is [21]

$$p_s(\gamma_i)=\frac{\binom{\gamma_i}{\bar{\gamma}_q}}{\binom{F}{\bar{\gamma}_q}}. \tag{10}$$

Therefore, by combining the previous two equations, the probability of a representative being activated is

$$p_{rep}(\gamma_i)\simeq p_s(\gamma_i)p_{act}(b) \tag{11}$$

and consequently, the probability of accessing a page that contains x b_groups with $\overline{\gamma_i}$ average s_descriptor weight is

$$p_{pg}(x,\overline{\gamma_i})=1-(1-p_{rep}(\overline{\gamma_i}))^x. \tag{12}$$

Assuming that the hash level is l and the number of partitions is PT, then the total number of page accesses needed to satisfy a query Q with weight γ_q is

$$\begin{aligned}DiskAcc(Q)=&(2PT-2^l)\cdot[2^{-\bar{\gamma}_q(l)}\cdot p_{act}(\bar{n}_{sb})\cdot(1+\bar{n}_{pb}\cdot p_{pg}(\bar{n}_{rp},\bar{\gamma}_i))]\\&+(2^l-PT)\cdot[2^{-\bar{\gamma}_q(l-1)}\cdot p_{act}(\bar{n}_{sb})*(1+\bar{n}_{pb}\cdot p_{pg}(\bar{n}_{rp},\bar{\gamma}_i))].\end{aligned} \tag{13}$$

Considering the second approach, the probability of a signature at depth d to contain 1s at γ_q prespecified positions is [20]

$$p(\gamma_q,d)=\left[1-\left(1-\frac{\gamma}{F}\right)^{\lambda(d)}\right]^{\gamma_q}. \tag{14}$$

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 equals:

$$\lambda(d)=\frac{n}{\Pi_{i=1}^{d}\bar{n}_i}, \tag{15}$$

where $\bar{n}_i$ is the average number of signatures per node at depth i.

In this study, the first level is produced by linear hashing which contributes *hashAcc* accesses as shown previously. Checking for matching signatures begins at the second level. By noticing that each representative is produced by the superimposition of $\bar{n}_4$ signatures of a b_group, where:

$$\bar{n}_4=\frac{\bar{n}_3}{\bar{n}_{rp}} \tag{16}$$

the probability of a signature to be activated at this level will be

$$p(\gamma_q, 2) = \left[1 - \left(1 - \frac{\gamma}{F}\right)^{\bar{n}_4}\right]^{\gamma_q} \tag{17}$$

and, therefore, $\bar{n}_2 \cdot p(\gamma_q, 2)$ signatures from each block will correspond to the query.

In order to derive the number of page accesses, one has to consider that the signatures which are stored in the second level might not belong to discrete pages, since (depending on the method used) a page could produce more than one representatives. In addition, considering also the uniform distribution of 1s, then the number of produced page accesses must be divided by $\bar{n}_{rp}$. Consequently, the expected number of page accesses will be

$$DiskAcc(Q) = hashAcc \cdot \frac{\bar{n}_2 \cdot p(\gamma_q, 2)}{\bar{n}_{rp}}. \tag{18}$$

6. Experimental results

A thorough experimental evaluation of all previously described methods has been contacted in order to compare their efficiency under varying parameters. The structures were implemented in C++ and the experiments run on a Pentium II workstation under Windows NT. All considered parameters and the corresponding tested values are given in Table 2. The performance measure considered was the number of disk accesses required to satisfy a query. For each query weight, an average of 100 measurements was taken.

In the following figures, with respect to the PWF method we have considered six b_groups per page (i.e. $b = 6$) for both cases of $F = 512$, $P = 2$K, and $F = 1024$, $P = 4$K, since this is its best tuning as concluded in [17].

6.1. Evaluation of estimation functions

In the first set of experiments we focused in evaluating the estimation functions which were described in the previous section. We experimented with each distinct method for a varying number of signature weights and we present the results of a representative part for both the theoretical and experimental evaluation. The acronyms, which are used in the figures below, are the following:

- **Exp** for experimental results,
- **SAppr** (S-tree based Approach) for the estimation based on the approach described in [20], and
- **PAppr** (PWF based Approach) for the estimation based on the approach described in [17].

In Figs. 6 and 7 we see the results of our evaluation when applied on the LHS, LOC and LOG method, respectively. The signature size is $F = 512$ bits with $\bar{\gamma} = 154$ average weight, whereas the database size is

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 (in bits)	$F = 512, 1024$
Probability of a bit position to be set	$p_1 = 1/4, 1/3.5, 1/3, 1/2.5, 1/2$
Page size (in KB)	$P = 2, 4$
Minimum page capacity (as percentage)	$k = 35\%$ of max page capacity

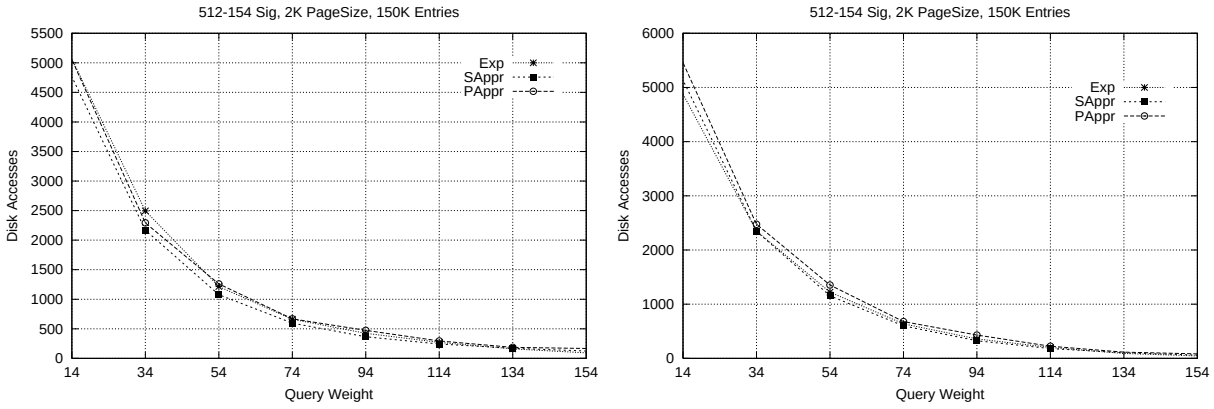

Fig. 6. Comparison of analytical estimates for the LHS (left) and LOC (right) methods as a function of the query weight.

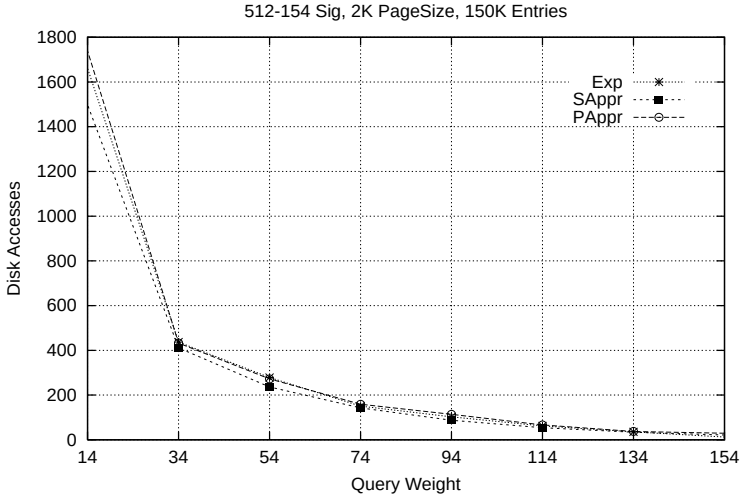

Fig. 7. Comparison of analytical estimates for the LOG method as a function of the query weight.

$n = 150,000$ entries. It can be clearly seen that the deviation between the analytical and the experimental results is quite reasonable for all proposed methods. Specifically, the deviation for all three methods, for both estimation methodologies ranges between 5% and 13% for query weights smaller than $\bar{\gamma}_q = 135$, while only for very large query weights it is over 20%. Therefore, both estimation formulae represent quite effectively the behavior of the methods except of very heavy query signatures. Similar results were obtained by using other parameter values.

6.2. Evaluation of the experimental results

6.2.1. Performance over different element weights

Before examining the performance of each method, a closer look should be taken on the parameters that were used for the experiments. Assuming that the database contains n objects, each object will have an indexed multi-valued attribute, whose value is a set of cardinality D. As described in Section 2.1, when D

signatures are superimposed, the minimum false drop probability can be achieved if the weight of each element, m, is such that the resulting probability of a bit position to be set after superimposition is $\frac{1}{2}$ (see Eq. (1)). However, experiments in [10] showed that if m is smaller than the optimum, a better retrieval performance is accomplished. Having this in mind and considering also the fact that the cardinality of a set-valued attribute is usually relatively small, we have tried to tune this parameter by experimenting with smaller values than the theoretical optimum.

Indeed, experiments were performed on subset queries in order to examine the performance of the five methods with respect to the number of false drops. We varied the number of attributes per set from 5 to 25, the cardinality of the attribute domain from 1000 to 100,000 items, the number of queried items from 1 to the number of items existing in the set, and we experimented with both a uniform and a skewed distributions. The results showed that the increase in false drop probability was 6% on the average, whereas it was 25% in the worst case (in skewed distributions).

On the other hand, querying the same number of items in all methods for different m values showed that by lowering m the retrieval cost was greatly improved. More specifically, in Fig. 8 we illustrate the performance of all methods for signatures of $F = 512$ size for increasing values of m, where each object contains 10 items and 5 items are queried. From this figure it is derived that all proposed methods outperform the PWF and S-tree structures (denoted as STR) for small m values, whereas the LOG method performs equally well with PWF even for larger m values. It should also be noticed that if m is small, then an 80–95% decrease in disk accesses can be achieved by using the LOC and LOG methods instead of using the PWF method with a higher value.

Finally, apart from the previous two experimental results, it must also be mentioned that when the weight of signatures increases, there is a dramatic increase in the weight of the produced representatives, prohibiting therefore the avoidance of certain unqualified paths since no filtering criteria can be applied. As a result, all methods tend to behave simply as storage structures and not as smart indexes, i.e. as a means that can help eliminate the traversal of unnecessary paths. PWF behaves the best in such cases except of the case of heavy query weights where the other methods still prevail. This behavior is due to the way signatures are packed in the PWF structure, which is tighter than in the remaining methods, creating this way a quite smaller number of leaves.

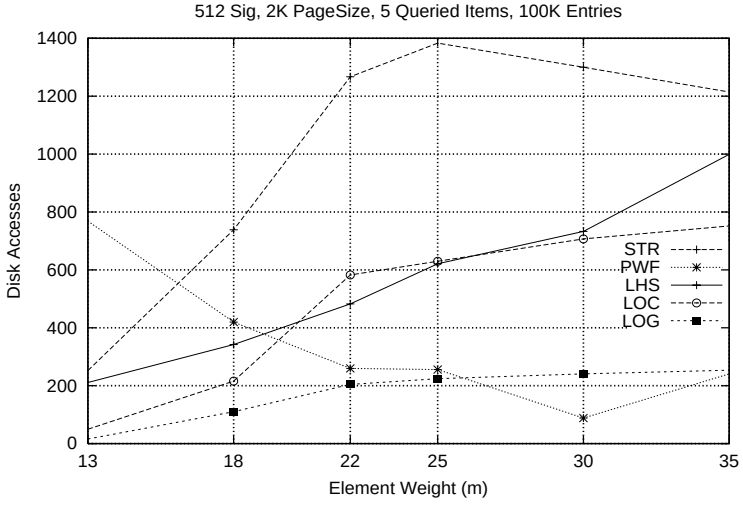

Fig. 8. Comparison of the proposed methods as a function of the weight of inserted signatures.

Having all previous observations in mind, we proceeded with the remaining experiments by using m values that would lead to the creation of object signatures with $p_1 = \frac{1}{3}$ or $p_1 = \frac{1}{4}$ on the average.

6.2.2. Tuning the LOG method

Experiments showed that when the number of logical pages per physical page was increasing, the storage overhead increased as well. This was the result of the increased number of splits (now less entries correspond to each logical page) and therefore the increased percentage of unused space left in each page. In Fig. 9 we show the results of the retrieval cost and the space overhead needed for the case of 1, 2, 3 and 4 logical pages per physical page in a physical page of 2K size. It should be noticed here that the LOG method with only one logical per physical page is identical with the LHS method. More specifically, as it is shown in the two parts of the figure, having 2 logical pages instead of 1 improves the performance by 77% at a 17% cost of space overhead. Similarly, when using 3 or 4 logical pages instead of only 2, the performance is improved by 82% and 92%, at a 8.5% and 16% average cost for storage space, respectively. However, although performance improves greatly by paying a comparatively small storage cost as far as partition pages are concerned, the hash table expands greatly, especially in the case of 3 or 4 logical pages, making it quite costly to maintain in main memory. For example, in the case of 1 or 2 logical pages the hash table produces around 1000 partition blocks while in the case of 3 or 4 logical pages it reaches 2000 and 3000 partition blocks, respectively. In addition, when the superset query is evaluated, both the cases of 3 and even worse the case of 4 logical pages per physical one, are very costly to use since in such a query the largest part of the structure has to be accessed. Considering the previous two observations, in the following, we examine only the LOG method with two logical pages per one physical.

6.2.3. Evaluation of the proposed methods over different query weights

In Fig. 10 the performance of the five different methods under examination is shown for signatures of $F = 512$ bits size stored in 2K pages. The average weight of the signatures is $\bar{\gamma} = 120$ (i.e. $p_1 \simeq \frac{1}{4}$) on the left and $\bar{\gamma} = 154$ (i.e. $p_1 \simeq \frac{1}{3}$) on the right part.

The LOG method behaves by far the best compared to the other methods, reaching even a 70–90% improvement when compared to the PWF or STR method, whereas LOC seems to attain a 50–80% improvement over PWF. Inferior performance is shown by the LHS method which is still remarkable, about 50–60% better than the PWF and STR methods. Similar results can be observed in Fig. 11, where the

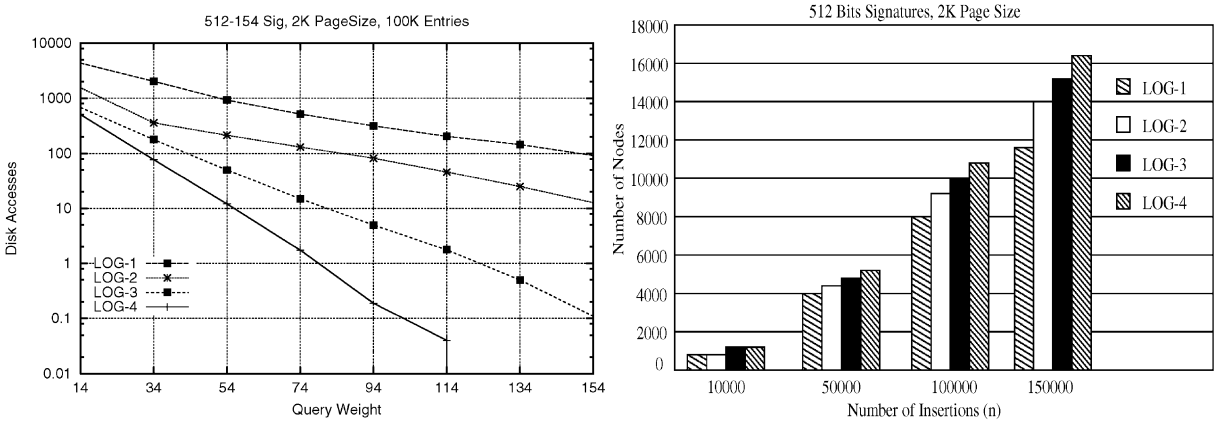

Fig. 9. Performance of the LOG method: retrieval costs (left) and storage overhead (right).

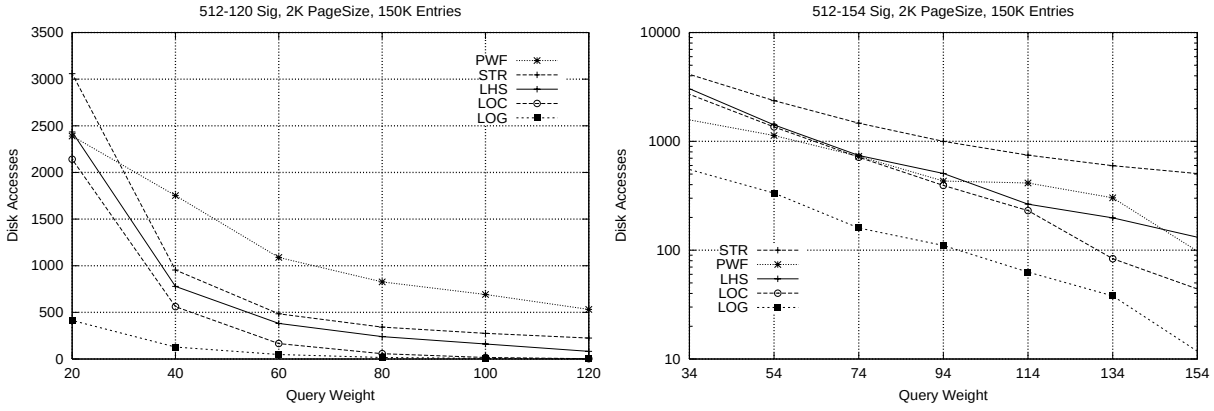

Fig. 10. Comparison of the proposed methods for 512–120 signatures (left) and 512–154 signatures (right) in 2K pages, as a function of the query weight.

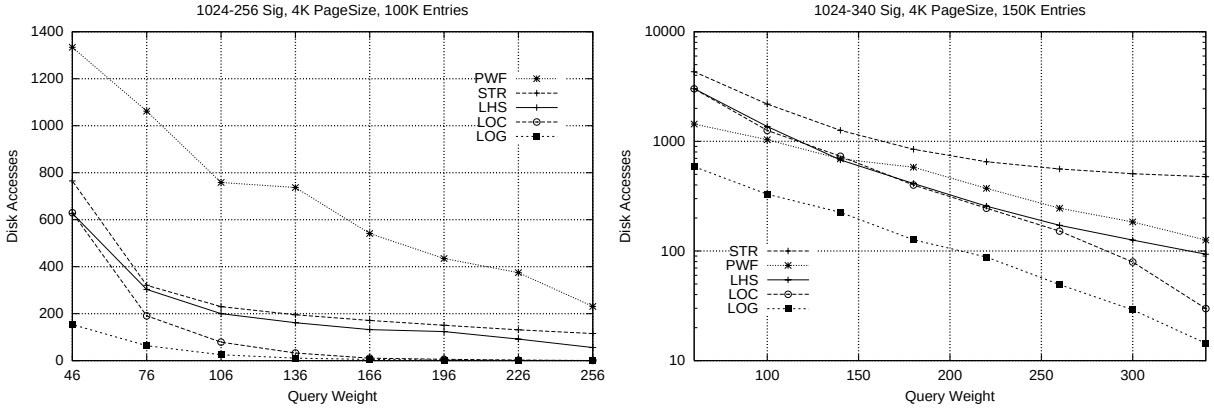

Fig. 11. Comparison of the proposed methods for 1024–256 signatures (left) and 1024–340 signatures (right) in 4K pages, as a function of the query weight.

performance of the methods for signatures of size $F = 1024$ bits in 4K pages is shown, with an average weight of $\bar{\gamma} = 256$ bits on the left and $\bar{\gamma} = 340$ bits on the right.

Apparently, the first conclusion that we reach is that without taking into consideration logical pages, the use of similarity measures by itself can contribute to a significant improvement of the storage costs as is shown by the performance of the LHS method. On the other hand, when logical pages are created as well, the performance is improved even more, since the number of signatures that are superimposed is decreased and the clustering achieved is much finer.

Although PWF is not the best method, it demonstrates a good performance in the case of very light queries. This can be explained by the fact that, in such light queries, all other methods tend to converge to the attitude of R-tree like structures, i.e. they cannot easily eliminate search paths, accessing this way a large number of leaves. PWF on the other hand, is very packed and creates fewer leaves which also make the difference in its performance.

Lastly, in Fig. 12, the time overhead for the most efficient of the methods is presented. It can be seen that, disk accesses are also translated to an improvement in time since a much lower number of leaves has to be

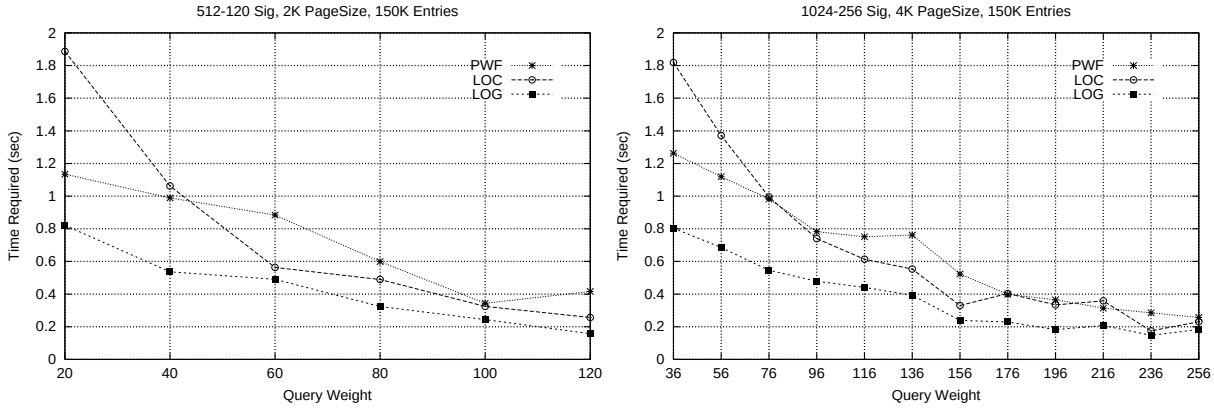

Fig. 12. Time overhead of the proposed methods for 512–120 signatures (left) and 1024–256 signatures (right) in 2K and 4K pages, respectively, as a function of the query weight.

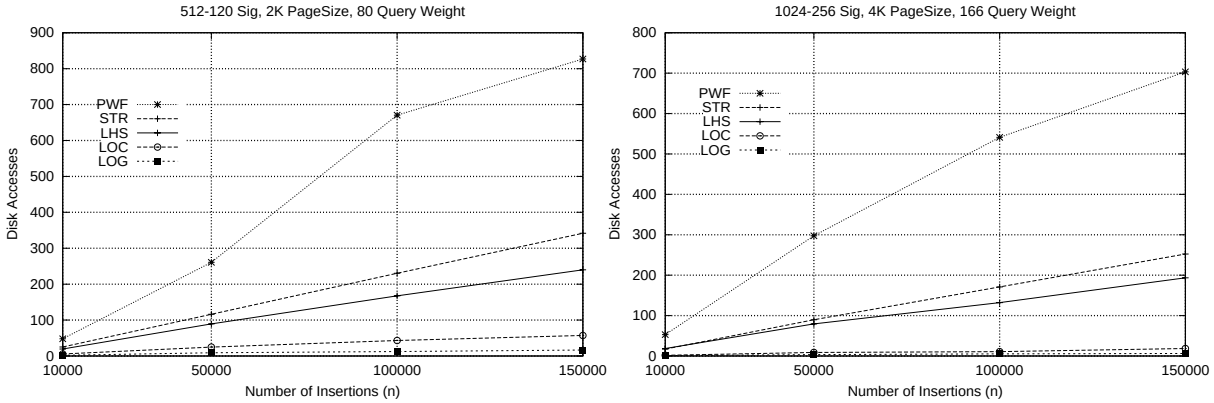

Fig. 13. Comparison of the proposed methods as a function of the number of inserted signatures for 512–120 signatures (left) and 1024–256 signatures (right).

read from secondary memory. However, the improvement is not proportional to the disk access reduction due to the sometimes increased number of signatures that has to be compared in order to evaluate a query. Specifically, a 30% and 42% decrease in time is noticed for the case of 512 signature size for the LOC and LOG method, respectively, while for the 1024 signature size the decrease observed was 27% and 40%. Therefore, even in time, the LOG method seems to outperform all the other methods under discussion.

6.2.4. *Performance over increasing number of entries*

Figs. 13 and 14, illustrate the disk accesses of all methods with respect to the number of stored signatures. In the left part, the signatures used are of $F = 512$ bits size with $\bar{\gamma} = 120$ and $\bar{\gamma} = 154$ average weight, whereas on the right part they are of $F = 1024$ bits size with $\bar{\gamma} = 256$ and $\bar{\gamma} = 340$ average weight, respectively. It is clear that all methods present a linear increase in the number of accesses required to answer a query having the LOG method preserve its best behavior.

However, there is a significant difference in performance as far as the PWF and STR methods are concerned. It is obvious that the performance of the S-tree is greatly affected by the average weight of the

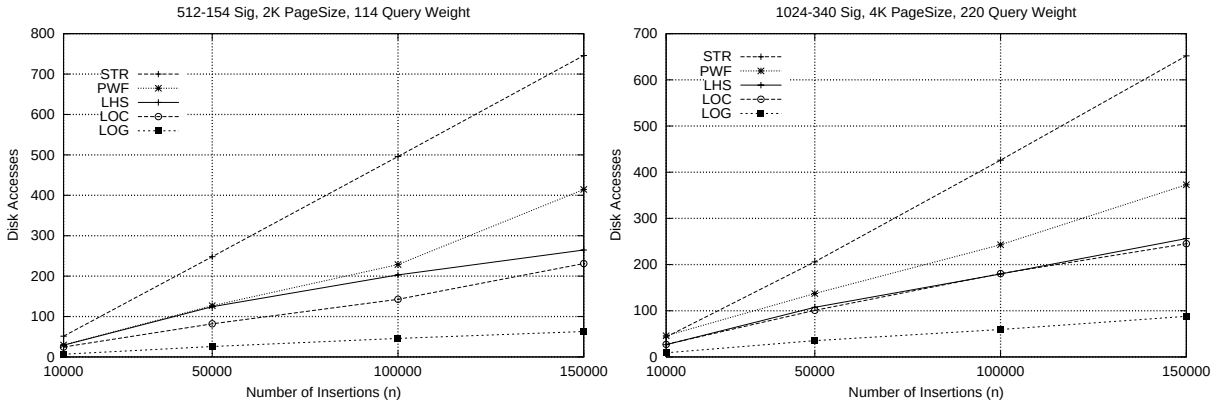

Fig. 14. Comparison of the proposed methods as a function of the number of inserted signatures for 512–154 signatures (left) and 1024–340 signatures (right).

object signature, and consequently by the m weight. Actually STR shows its best behavior for lower values of m, whereas PWF for higher ones, for reasons explained earlier in the section.

6.2.5. *Proposed methods over superset queries*

The last group of experiments involved testing of the methods for superset queries. As it is generally accepted, due to the fact that the S-tree cannot answer this query unless almost all leaves are retrieved, it makes it prohibitively costly for use in such queries and therefore, we did not include it in our experiments. As far as the remaining methods are concerned, we used implementations of all methods with l_descriptors added to the representative of each group and to hash table entries for every partition block. As mentioned in Section 3, PWF does not answer superset queries. Thus, we also expanded it in this way in order to test its behavior.

With these modifications, when a superset query is issued, we use the l_descriptors instead of m_descriptors. Recall that the l_descriptor contains the minimum number of 1s existing in each $\lfloor \log F \rfloor$ part of a group or a block. Therefore, in case the queried signature has l_descriptor 'smaller' than the one of a block or a group, the corresponding branch can be skipped since the superset condition cannot be satisfied.

In Fig. 15 we show the results of a superset query when applied on signatures of size $F = 512$ bits containing 5 elements in their set-valued attribute, while the number of queried items varies from 5 to 50. As it can be seen, the methods that seem to prevail are the PWF and LHS ones, with the LHS showing an improvement of 20% on the average over the PWF method. This can be explained by the fact that the superset query can only be restricted by the hash key and the l_descriptor and consequently, a largest part of the index is retrieved. As a result, the methods that seem to prevail are the ones that achieve a limited number of partition pages in combination with a more refined hash key and l_descriptor.

6.2.6. *Storage overhead*

In the left part of Fig. 16, we illustrate the required storage space when signatures of $F = 512$ size are stored in 2K pages. From this figure it is obvious that PWF needs the least space since it is the mostly packed method. On the other hand, the most overhead is presented by the LOG method because, due to the independent nature of its logical pages, it pays the cost of creating nodes with low load factor. LOC, LHS and STR methods behave pretty similarly together and comparatively well to the PWF method.

However, as it can be seen from the right part of the same figure, the number of overflow pages that are created is increased by 37% approximately in the case of PWF, whereas for the rest of the methods is much

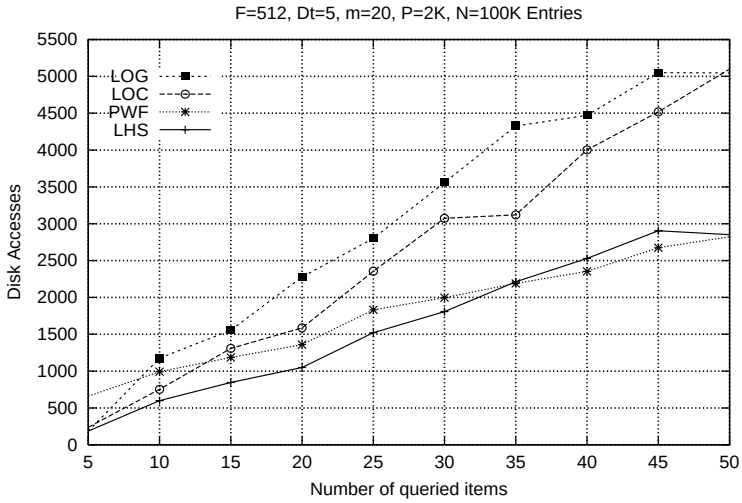

Fig. 15. Comparison of the proposed methods over a superset query for 512 signatures.

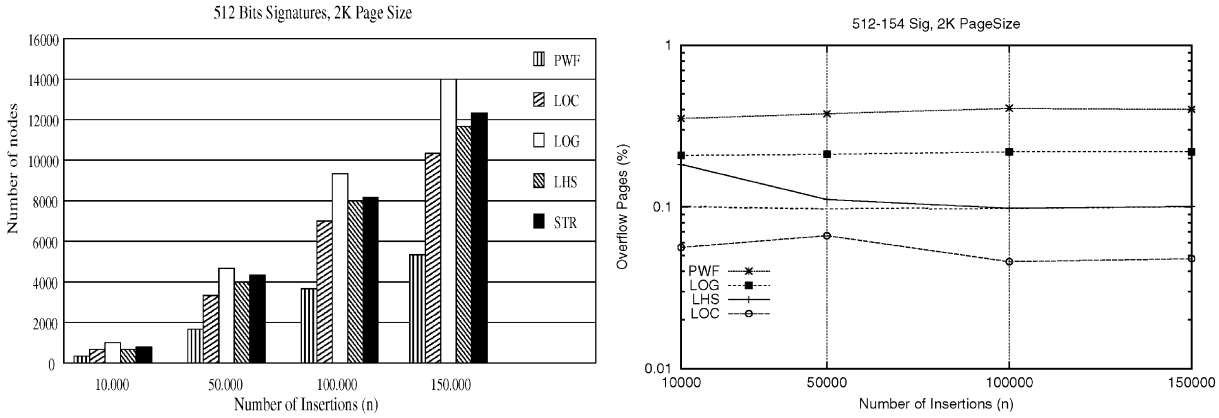

Fig. 16. Storage overhead (left) and the percentage of created overflow pages (right) of the five methods for 512 bits signatures and 2K page size.

lower. Specifically, methods LHS, LOC and LOG present an overflow of 15%, 5% and 21% on the average, respectively. The difference in percentage is due to the fact that, since LHS and LOC methods are not packed, overflows happen more often, leading to the expansion of the hash table and to the concurrent reorganization of overflow pages. The adverse situation happens in PWF and LOG, which both perform some kind of packing. Especially, the PWF method presents the most overflow space due to its small hash table. STR is not mentioned in this figure since it creates no overflow pages.

7. Conclusion

Inclusive (i.e. subset and superset) queries are often posed in object-oriented and object-relational databases, where objects have set-valued attributes. There has been very limited work in the literature with

regards to indexing sets. In several object-oriented systems, like O_2, GemStone and Orion there is no support for subset and superset queries; some of them provide a set type constructor along with a number of operations such as: insert/delete an object in/from a set, return the set's cardinal number (cardinality), test whether an object belongs to a set (membership), and test whether a set is a subset/superset of another set, whereas ObjectStore provides a kind of subset query to objects of class *Collection*. The same observation can also be made for the object-relational database systems. Thus, the present paper aims at the establishment of new efficient access methods tailored to index and query sets.

For the above purpose, we have based our approach on two earlier access methods: linear Hashing and S-trees. Moreover, we have extended a recent structure, the parametric weighted filter, and we have introduced three variations of a new hybrid scheme. We have shown experimentally how each one of the three variations combines the advantages and deals with shortcomings and decision issues that stem from each one of the original methods. In simple words, due to improved clustering, the new methods outperform S-trees and the parametric weighted filter in almost all cases of our thorough experimentation in the case of subset queries. Also, in the case of superset queries, we have improved by far the performance of the S-tree which needs to search almost the whole structure in order to evaluate such a query, while we have also improved the performance of the PWF by expanding the auxiliary information it maintains.

In addition, by following two methodologies, for each one of the three new methods we have derived analytical formulae to describe the method's performance (in disk accesses). Experimentally, it is also shown that the derived formulae provide close approximations (with low computational costs) in comparison to the experimental results. Thus, these analytical functions could be of use in a query evaluation engine.

Future work could involve the application of the previous methods in a distributed environment — like the gaining popularity SSDS model [28] — or in a parallel environment [29,30]. Finally, another step for future research is the examination of the method's performance on joining sets [23,26].

Appendix

As mentioned in Section 4.1, there could be a case where a signature does not belong to the appropriate partition but to one of its siblings. In the following, it is explained why this is a legal case and how, during a search, all qualifying signatures will still be retrieved.

In order to clarify the situation, in Fig. 17 a LHS structure is depicted where the level of the hash table is 1, and consequently the only existing partitions are those ending in a '0' and a '1' suffix. All signatures stored in the first partition have in common only the last bit which is the '0' suffix of the corresponding hash table entry. Therefore, there are three possible scenarios over the suffix of signatures stored in each page: either they all end in '00', producing a representative ending in '00', or they all end in '10' producing a representative ending in '10', or some of the signatures end in '00' and others in '10', still producing a representative that would end in '10' as a result of superimposition. In the last case, the fact that in the specific page exist signatures with a '00' suffix will be hidden.

Upon expansion of the hash table, the first partition to be split would be the '0' one, resulting therefore in the creation of the two new partitions with hash level 2 and with '00', '10' suffixes. As described in Section 4.1, this would cause the partition block to be split where the representatives ending in '00' would be stored in the first partition while those ending in '10' would be stored in the latter.

Let us see an example. The hash table of Fig. 17 is expanded and the '0' partition has to be split. Therefore, all representatives stored in this partition along with the corresponding pages will now be redistributed among the two new partitions according to their new 2-bit suffix, as shown in Fig. 18.

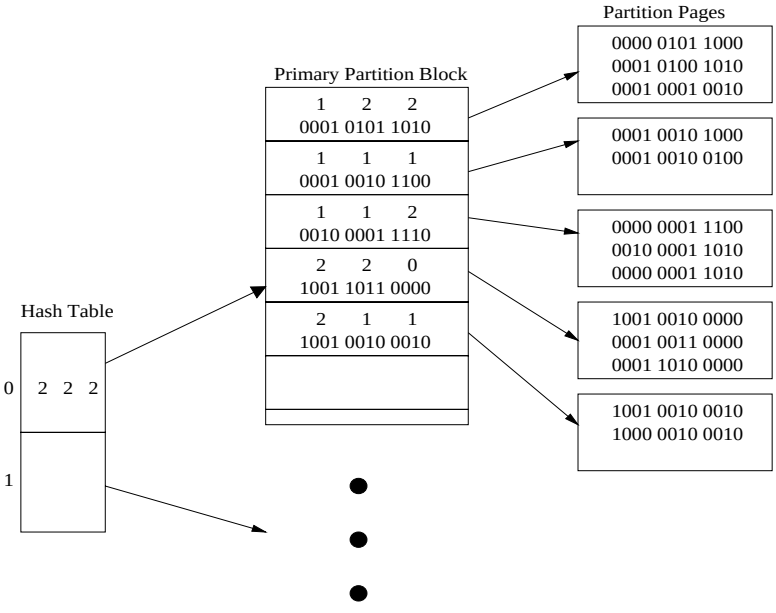

Fig. 17. An example of LHS with $F = 12$ and $h = 1$.

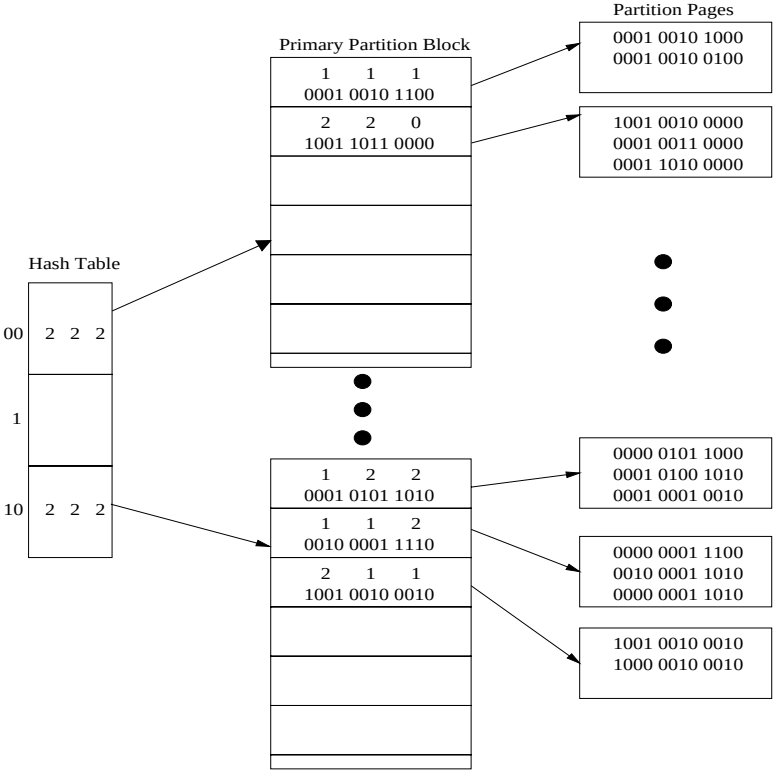

Fig. 18. The LHS structure of Fig. 16, after the hash table has been expanded.

Therefore, the second and fourth page will now belong to partition '00', while the rest of them will belong to the '10' partition. However, the 0000 0101 1000 signature, which is stored in the first page of the '0' partition, will now belong to the second partition and not to the '00' one as expected.

In spite of this fact though, searching will still be efficient. For example, upon a subset query, where the query signature is 0000 0100 1000, both '00' and '10' partitions will be visited resulting in the 0000 0101 1000 signature being evaluated.

References

[1] E. Bertino, B. Catania, L. Chiesa, Definition and analysis of index organizations for object-oriented database systems, Inf. Systems 23 (2) (1998) 65–108.
[2] E. Bertino, B.C. Ooi, R. Sacks-Davis, K.L. Tan, J. Zobel, B. Shidlovsky, B. Catania, Indexing Techniques for Advanced Database Systems, Kluwer Academic Publishers, Dordrecht, 1997.
[3] A. Kemper, G. Moerkotte, Access support relations: an indexing method for object bases, Inf. Systems 17 (2) (1992) 117–146.
[4] T.A. Mueck, M.L. Polaschek, Index data structures in object-oriented databases, Kluwer Academic Publishers, Dordrecht, 1997.
[5] Z. Xie, J. Han, Join index hierarchies for supporting efficient navigation in object-oriented databases, Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, pp. 522–533.
[6] S. Christodoulakis, C. Faloutsos, Signature files: an access method for documents and its analytical performance evaluation, ACM Trans. Office Inf. Systems 2 (4) (1984) 267–288.
[7] C. Faloutsos, Signature files, in: W.B. Frakes, R. Baeza-Yates (Eds.), Information Retrieval: Data Structures and Algorithms, Prentice-Hall, Englewood Cliffs, NJ, 1992.
[8] D. Dervos, Y. Manolopoulos, P. Linardis, Comparison of signature file models with superimposed coding, Inf. Process. Lett. 65 (2) (1998) 101–106.
[9] S. Kocberber, F. Can, J. Paton, Optimization of signature file parameters with varying record lengths, Comput. J. 42 (1) (1999) 11–23.
[10] Y. Ishikawa, H. Kitagawa, N. Ohbo, Evaluation of signature files as set access facilities in OODBs, Proceedings of the 1993 ACM SIGMOD Conference, Washington, DC, 1993, pp. 247–256.
[11] H. Kitagawa, Y. Fukushima, Y. Ishikawa, N. Ohbo, Estimation of false drops in set-valued object retrieval with signature files, Proceedings of the Fourth FODO Conference, Chicago, IL, 1993, pp. 146–163.
[12] H. Kitagawa, N. Watanabe, Y. Ishikawa, Design and evaluation of signature file organization incorporating vertical and horizontal decomposition schemes, Proceedings of the Seventh DEXA Conference, Zurich, Switzerland, 1996, pp. 875–888.
[13] P. Ciaccia, P. Tiberio, P. Zezula, Declustering of key-based partitioned signature files, ACM Trans. Database Systems 21 (3) (1996) 295–338.
[14] D.L. Lee, C.W. Leng, Partitioned signature files: design issues and performance evaluation, ACM Trans. Office Inf. Systems 7 (2) (1989) 158–180.
[15] D.L. Lee, C.W. Leng, A partitioned signature file structure for multiattribute and text retrieval, Proceedings of the Sixth ICDE Conference, Los Angeles, CA, 1990, pp. 389–397.
[16] P. Zezula, F. Rabitti, P. Tiberio, Dynamic partitioning of signature files, ACM Trans. Inf. Systems 9 (4) (1991) 336–369.
[17] P. Bozanis, C. Makris, A. Tsakalidis, Parametric weighted filter: an efficient dynamic manipulation of signature files, Comput. J. 38 (6) (1995) 478–488.
[18] R. Sacks-Davis, K. Ramamohanarao, A two level superimposed coding scheme for partial match retrieval, Inf. Systems 8 (4) (1983) 273–289.
[19] J. Pfaltz, W. Berman, E. Cagley, Partial match retrieval using indexed descriptor files, Commun. ACM 23 (9) (1980) 522–528.
[20] U. Deppisch, S-tree: a dynamic balanced signature index for office retrieval, Proceedings of the Ninth ACM SIGIR Conference, Pisa, Italy, 1986, pp. 77–87.
[21] E. Tousidou, A. Nanopoulos, Y. Manolopoulos, Improved methods for signature tree construction, Comput. J. 43 (4) (2000) 301–314.
[22] J.M. Hellerstein, A. Pfeffer, The RD-tree: an index structure for sets, Technical Report No. 1252, University of Wisconsin at Madison, 1994.
[23] S. Helmer, G. Moerkotte, Evaluation of main memory join algorithms for joins with set comparison join predicates, Proceedings of the 23rd VLDB Conference, Athens, Greece, 1997, pp. 386–395.
[24] W.C. Lee, D.L. Lee, Signature file methods for indexing object-oriented database systems, Proceedings of the Second Computer Science Conference, Hong Kong, 1992, pp. 616–622.

[25] H.S. Yong, S. Lee, H.J. Kim, Applying signatures for forward traversal query processing in object-oriented databases, Proceedings of the 10th ICDE Conference, Houston, Texas, 1994, pp. 518–525.
[26] K. Ramasamy, J.M. Patel, J.F. Naughton, R. Kaushik, Set containment joins: the good, the bad and the ugly, Proceedings of the 26th VLDB Conference, Cairo, Egypt, 2000, pp. 351–362.
[27] W. Litwin, Linear Hashing: a new tool for files and table addressing, Proceedings of the Sixth VLDB Conference, Montreal, Canada, 1980, pp. 212–223.
[28] W. Litwin, M.A. Neitman, D.A. Schneider, LH* — linear hashing for distributed files, ACM Trans. Database Systems 21 (4) (1996) 480–525.
[29] F. Grandi, P. Tiberio, P. Zezula, Frame-sliced partitioned parallel signature files, Proceedings of the 15th ACM SIGIR Conference, Copenhagen, Denmark, 1992, pp. 286–297.
[30] E. Tousidou, M. Vassilakopoulos, Y. Manolopoulos, Performance evaluation of parallel S-trees, J. Database Manage. 11 (3) (2000) 28–34.