Compressing Large Signature Trees

Maria Kontaki, Yannis Manolopoulos, and Alexandros Nanopoulos

Department of Informatics, Aristotle University of Thessaloniki, Greece
{kontaki,manolopo,alex}@delab.csd.auth.gr

Abstract. In this paper we present a new compression scheme for signature tree structures. Beyond the reduction of storage space, compression attains significant savings in terms of query processing. The latter issue is of critical importance when considering large collections of set valued data, e.g., in object-relational databases, where signature tree structures find important applications. The proposed scheme works on a per node basis, by reorganizing node entries according to their similarity, which results to sparse bit vectors that can be drastically compressed. Experimental results illustrate the efficiency gains due to the proposed scheme, especially for interesting real-world cases, like basket-market data or Web-server logs.

1 Introduction

Nowadays, the database sizes continuously increase due to the increase in the size of data. During the previous years, various structures have been proposed for the storage of data in smaller space, but mainly for their more efficient processing [WMB99]. One of these structures is the *signature tree* (S-tree) [Depp86], which is used to index objects with multi-valued attributes. Objects of this type are used in object-oriented databases, in digital library systems, in WWW search engines, or in multimedia databases.

The value of each attribute of an object can be represented by a signature, that is, a bit vector produced by applying a hashing function on the attribute's value. The total number of aces (bits equal to one) is the weight of the signature. An object's signature is produced by superimposing (i.e., OR-ing) each of its attribute signatures. Initially, signatures were used as indices in the signature files that are sequential structures and require the scanning of the entire collection of signatures.

Deppisch [Depp86] proposed the S-tree, which similar to the B+-tree, is a height-balanced tree structure. In the S-tree, a signature of a node at level i is produced by superimposing all signatures of its child nodes that at level i+1, considering that the root is at level 0. As result, the upper levels have signatures with 'heavy' weight (many aces with respect to the length of signature). Therefore, the selectivity of such nodes reduces and the performance of structure during query processing is affected, since for each query a large number of nodes are retrieved. With primary goal the reduction of signatures weight, improved signature structures were proposed [TNM00].

Due to the way that signature construction is performed (using a hashing function), information loss may be imported: it is possible that two different objects have the

L. Kalinichenko et al. (Eds.): ADBIS 2003, LNCS 2798, pp. 163–177, 2003.

same signature and, thus, it is possible to retrieve objects that do not satisfy the query. The latter case corresponds to the *false-drops*. False-drops affect the performance of the structure, and for this reason, several methods of signature construction were developed, which decrease the false-drops probability [Zezu88].

Two factors mainly affect the performance: (a) the node retrieval time (I/O) and (b) the node processing time (CPU). Although progress has been marked in the disk technology and the reduction of access time, it is, however, not as significant as the progress in processor speeds. Thus, factor (a) still has a significant impact. To overcome this problem, the technique of *compression* can be considered as means to decrease the number of disk accesses. Compression has been primarily associated with the reduction of storage space. However, nowadays, this does not anymore comprise a crucial objective, since the problem of storage space is not considered as much intense as the need of performance during query processing. For this reason, the focus in on how compression can reduce the I/O overhead (node retrieval time) by storing the nodes in less disk pages and, thus, reducing their retrieval time. By achieving a good rate of compression, the performance of query processing can be significantly improved, despite the overhead that is added by the additional (CPU) time required by the compression and decompression.

A generalized framework for compressing index structures (e.g. B-tree, R-tree, etc) has been introduced by [Teuh01], which describes two categories of compression: (a) compression of stored information, and (b) compression of pointers in nodes. The effectiveness of the former (a) category depends on the distribution and the number of different values. In this category compression can apply the lossy scheme (which leads to an increase of false drops) or the lossless scheme.

1.1 Contributions and Layout

In this paper, we propose a novel compression scheme for the S-tree, which is based on the aforementioned issues. The proposed scheme is lossless and is applied in the data entries of nodes (stored information), i.e., the (a) category according to the aforementioned framework. In particular, the scheme works on a per node basis by reorganizing node entries according to their similarity (exploiting node clustering), which results to sparse bit vectors that can be drastically compressed. Emphasis is given to the query processing performance. For this reason the scheme contains an efficient decompression method that requires low CPU times, thus it does not compromise the gains due to the reduction of I/O overhead. Moreover, the proposed scheme is based on compression techniques for sparse vectors, which have been studied in depth during the previous years [BK91].

The contributions of this paper are summarized in the following:

- The development of the compression scheme for the S-tree structure, according to the framework of [Teuh01].
- The development of a novel decompression method which, during query processing, avoids the decompression of the entire tree and, moreover, it uses optimisations (e.g., the adjustment of bits in the query according to the bits in the decompressed nodes). As a result, the CPU time required for decompression is kept significantly low.

- Detailed experiment results, with both real and synthetic data, which examine the weight of signatures, the correlation between the signatures (a case that is met often in real-world applications), and the query length.

It must be additionally noticed that although the proposed compression scheme follows the framework described in [Teuh01], the latter describes index structures in general (containing a small discussion on the S-tree, which comprises the motivation in our scheme), whereas its experiments focused on the R-tree. Our scheme proposes a number of significant contributions compared to the aforementioned general framework, namely: (a) the efficient decompression method, that makes feasible the query processing with the compressed tree by drastically reducing the CPU overhead; (b) the detailed examination of specific details with respect to signature data and the S-tree, e.g., the correlation between signatures; and (c) the experimental results that study the effectiveness of the approach on the S-tree structure.

The rest of the paper is organised as follows. Section 2 describes the related work. In Section 3, we present the proposed scheme. Section 4 contains the experimental results, while the conclusions and the future work are given in Section 5.

2 Related Work

The first approach to index signatures was through sequential signature files [CF84]. Although sequential files reduce the cost for searching the data, they have the drawback that all the signatures in the sequential file are probed during the searching.
S-trees have been proposed in [Depp86], to overcome the aforementioned problem. S-trees are height-balanced tree (analogous to B^+-trees) and they organize the signatures according to criteria like the minimization of weight increase (for more details, see Section 2.1). Also, [S-DR87] have proposed a two-level index structure for the efficient organization of signatures.

Several shortcomings of the original S-tree, especially with respect to the node-split policy, where addressed in [TNM00]. Other improvements for the organization of signatures in tree structures can be found in [TBM02,NM02], which also examine different types of queries (e.g., super-set queries, similarity queries, etc). Extensions of the use of signature indexes to several applications are included in [NTVM02, MNT03]

A description of compression schemes that can be applied to tree structures, in general, is given in the [Teuh01]. Experiments in [Teuh01] have focused on the R-tree structure, whereas it also contains a small description regarding the S-tree, which forms the motivation in our approach (for the new contributions of our approach, see the discussion in Section 1.1). A thorough examination of the advantages of compression can be found in the [WMB99]. Finally, [BBJK+00] proposes a compression method for indexes for high dimensional spaces and [GRS98] for relational databases and indexes for relational data. However, these approaches address much different requirements than the ones considered by our approach.

2.1 The S-Tree

S-trees [Depp86], similarly to B-trees, are height-balanced trees having all leaves at the same level. 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: $1 \leq k \leq K/2$. 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.

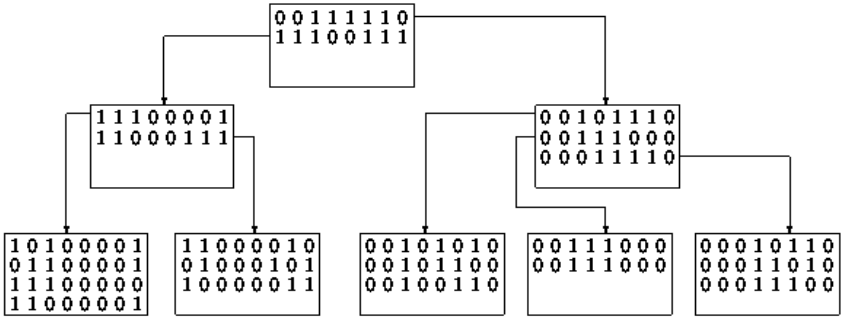

Fig. 1. An example of an S-tree

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. Due to lack of space, more details (regarding, insertion, deletion, etc.) can be found in [Depp86].

3 Proposed Scheme

Based on the general framework of [Teuh01], we consider two approaches of compressing an entry x of node N: (a) compressing x with respect to the content of the father of node N, and (b) compressing x with respect to the entries that are stored before x in the node N. Both (a) and (b) cases stand exceptions, without of course creating problem in the compression. For instance, in case (a), the root does not have father node, and, in case (b), the first signature of each node does not have any entries that are stored prior to it. Since the use of (a) does not prevent the use of (b), and vice versa, the proposed scheme exploits both of them. The gain from the first approach (a) is that it limits the possible values of x, thus x is stored in a more condensed way. The second approach (b) represents x as $px+\Delta x$, where px it is the value of the previous entry in N and Δx is the difference between x and px. The difference Δx is possible to contain less information than the entry itself, especially when considering the existence of similarity between the entries stored in a node; something that is in general attained by tree index structures, like the S-tree. Therefore, the gain is due to the smaller requirements for the storage space of Δx compared to that of x.

Regarding query execution, we focus on *containment* queries. In terms of signatures, given a query signature q, such queries search for those signatures s in the tree for which it holds that s AND $q = q$ (these queries are also called *subset* queries). Evidently, for the purpose of query processing, a straightforward approach is to first decompress the entire S-tree and then to execute the query. Nevertheless, we develop a different method, which avoids the cost of decompressing the entire tree and concentrates only to the relevant parts of it (i.e., those invoked by the query). In the remainder of this section, we describe in more detail the compression/decompression methods.

3.1 Compression Method

The first step of the method is the compression with respect to the father node. The signature of the father has resulted by applying the OR-ing of the signatures in its children nodes, and thus it is impossible to have any signature in a child node that has ace in a position that the signature of father has zero. Consequently all these zeroes, which "are imposed" by the father, can be omitted.

As described, the second step of the method is the compression with respect to the previous entry in the same node. This step required that consecutive signatures in a node to be as much as possible similar. As a distance measure (i.e., measure of inverse similarity) one can use the broadly used *hamming* distance, that is, the number of bits that the signatures differ. Since the signatures in a node of the S-tree are not ordered, and they are entered in the order they arrive, we can order the signatures within the node so that the total sum of distance between consecutive signatures is minimized.

The aforementioned requirement can be easily transmuted in finding a solution for the problems that belong in the family of travelling salesman problems (TSP). TSP is applied to an underlying graph. In our case, the graph consists of the node's signatures, which correspond to the vertices, whereas the edges are the intelligible lines that imply consecutive signatures. The hamming distance between two signatures

(vertices) comprises the weight of the corresponding edge that connects them. Evidently, for large graphs (that correspond to nodes with many entries), one has to resort to one of the several well-known heuristics for the TSP problem. For purposes of simplicity, we used the heuristic that is based on the minimum spanning tree of the graph.

Considering all the aforementioned issues, the compression scheme for a given node, is described as follows:

1. Remove from signatures of the children nodes, the positions that the signature of the father node has zeroes (obvious all the signatures of root are excluded from this step).
2. Apply a heuristic for the travelling salesman problem. The result of the heuristic will be an ordering of the node's signatures, which results to a small sum of Hamming distances between the consecutive signatures within this ordering.
3. Store the first signature of the resulting ordering, as it is. For each of the following signatures, calculate the XOR result with its previous signature (by applying the XOR logical operator between the bits in corresponding positions).
4. The signatures that result from step 3 are sparse vectors (due to the minimization of hamming distance). Apply a compression of the resulting sparse vectors (to be explained in the following).

To perform the compression in the entire S-tree, we have to apply the previously described algorithm (that works on a per node basis) to each of its node. The corresponding algorithm must be applied in a post-order tree traversal. This way of traversal is required due to the need of knowing the bits of the father's signature that are equal to 0, when compressing the signatures in its children. Initially, the root node is excluded from the compression. Then, for each node, we apply the node compression to its children nodes, store them, and then apply it also to the node itself. For a more clear description, the 3rd step of the node compression method is exemplified in Fig. 2.

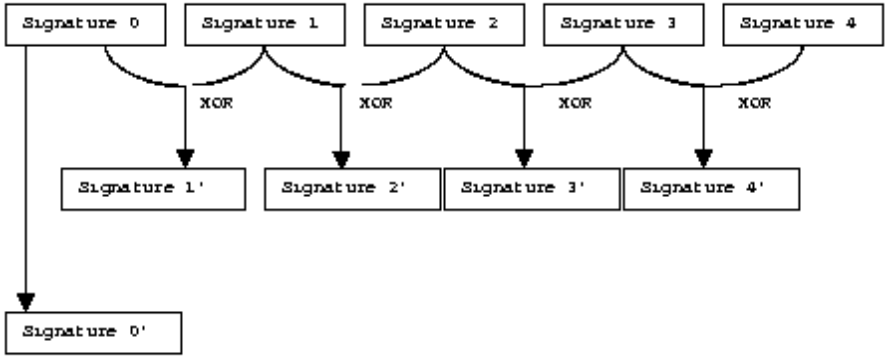

Fig. 2. An example of step 3 in the S-tree node compression scheme

For the 4th step of the node compression method, in our implementation we used a simple way of compressing sparse vectors. Consider that l is the length of signature and s is the number of aces in the signature (i.e., the weight of signature). For sparse vectors it holds that $s << l$. We mark the bit positions that are equal to *1*. For the storage of a position, $d=|\log_2 l|$ bits are required. Therefore, for each signature, $s \times d$ bits are needed. Evidently, the use of more sophisticated methods for compressing sparse vectors will lead to improved compression of the entire structure. This is left as a topic of future work, since herein we are interested in testing the effectiveness of the proposed scheme regardless of the specific method used for sparse vectors (i.e., we would like test its viability even for simple such methods). Important is the observation that, with respect to the proposed compression scheme, the resulting signatures do not have the same length (even in the same node), due to the compression of sparse vectors. Therefore, along with each sparse vector, we also store the length of the resulting signature (in Step 4 of the node compression method).

Finally, we have to notice that, in the uncompressed S-tree, each node is stored in an entire disk page, something that will lead to space wastefulness in the case of the compressed S-tree. The reason is that a compressed node occupies smaller space compared to an uncompressed one. Therefore, at each disk page, we store as many (compressed) nodes as possible. In our implementation, for reasons of simplicity, we considered that a node does not span different pages, thus a small fragmentation may incur. Nevertheless, if we do not use this simplification, the performance gains are expected to increase further.

3.2 Basic Node Decompression Method

As described, a straightforward method for processing queries would be to apply the reverse process of compression for the entire tree and, then, to execute the query over the decompressed tree. However, this would not be effective for two reasons: (a) no gain can be reaped, because the latter part in this procedure is equivalent to the querying of the tree when compression is not applied at all; (b) the total cost is burdened by the additional cost to uncompress the tree. Obviously, due to (a) and (b), the total performance would be worse than in the case where compression is not used.

Thus, in the proposed scheme, only the required part of the tree (the nodes invoked by the query) is involved during decompression, which is performed simultaneously with the processing of the query. Therefore, the overhead of decompression is decreased drastically.

As in the compression procedure, herein we will first describe a node decompression method and then the complete decompression algorithm. The first step in the node decompression method is the decompression of sparse vectors: we compute the number d (number of the bits that is required for the storage of a bit with value equal to one in an uncompressed signature) and, moreover, we initialize the new signature (all bits are initially set to 0). We divide the signature into groups of d bits. We find the positions of aces and set them to 1 (putting 1 to the corresponding bit of the new signature), turning each such group into a decimal number. It has been observed (by our experiments) that this is the most time-consuming step in the whole decompression method.

The next step of the node decompression method is to "inverse" the XOR operation between the consecutive signatures. We begin from most leftmost (first) signature of the node, in which the XOR operation was not applied so as to constitute our base for the inversion. For each pair of consecutive signatures, we apply the XOR operation (notice that by XOR-ing twice, we get the inverse) and, thus, we take the initial signatures. In Figure 3 is given an example that clarifies the previous procedure.

Finally, the node decompression method is completed by adding to the node's signatures the bits of the father's signature, which are equal to 0.

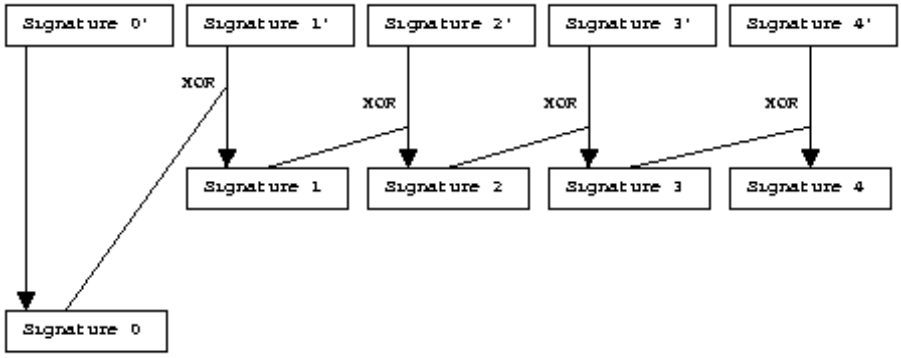

Fig. 3. An example of node decompression

3.3 Improved Decompression Method

In the basic node decompression method, the last step (i.e., the addition of bits according to the ones in the father's node signature, which have value equal to 0), overloads by far the decompression time, since it has to be applied in each signature of the node. To overcome this problem, we can omit the corresponding bits of the father's signature (i.e., those equal to 0) from the query signature. Please recall that the decompression is executed at the same time with the query. For example, let a node containing 90 signatures and its father's signature has 15 bits equal to 0. Therefore with the basic process, we should add 1350 bits, in total, to the signatures of the node. In contrast, with the method that was previously described, we entirely avoid this cost. It has to be noticed that the omission of the bits from the query signature is performed only for the signatures in the subtree of the father's signature (i.e., at each node, a local copy of the query's signature is used). From this, it is easy to see that the correctness of the query result is not affected.

Regarding the complete decompression method, different from the case of compression, the node decompression algorithm is applied through a *preorder* traversal of the S-tree. This is because it is necessary to first decompress the father-node before we continue with his children, so as to know the bit positions that are equal to 0. The

decompression method considers as basic unit the node (i.e. when one node is selected, all his signatures decompress). This is contrast to the approach of [GRS98], which considers as basic unit each individual entry in an index structure.

4 Performance Study

In this section, we present the experiment results that measure the performance gains due to the proposed scheme. We have conducted experiments to measure the compression rate (i.e., ratio of size between the compressed and uncompressed cases). However, due to space constraints, we herein focus on the measurement of performance during query execution, since, as described, this measure corresponds to the case of interest. The compressed S-tree is denoted as COMP and the uncompressed as ORIG. Next, we first describe the experimental setup and then we present the results.

4.1 Experimental Setup

For both COMP and ORIG we used the improved construction methods that were proposed in [TNM00]. For each experiment we present two diagrams: (a) one for the comparison between COMP and ORIG in terms of the disk accesses required by the query, and (b) one for the CPU time (measured in seconds) required by the decompression during the query execution (this time is presented only for COMP, since ORIG does not require it). The number of the disk accesses does not result from the count of the actual disk accesses but from the count of disk pages accesses and this holds for all experiments. As mentioned, we focus on the containment (subset) queries, which are measured with respect to the items involved in the query signature. The default page size is 8 K. The experiments were conducted on a PC with processor AMD Athlon at 1.6 GHz.

We used both real and synthetic data sets, with various values of weight/length of signatures. Hereafter, s denotes a signature, F its length, and with $w(s)$ its weight (the number of aces that contains). In addition, we use the notion of *fraction weight*, denoted as $wF(s)$, in order to express the number of bits in s that are equal to 1 with respect to its length (e.g., if F is equal to 512 and $w(s)$ is 256, then $wF(s)$ is equal to 0.5). For synthetic data, a fraction of the number of bits that were equal to 1 in a signature were also set to 1 in the immediately next generated signature. This yields to a correlation between the generated signatures. We have observed this in several real-world cases (e.g., basket market data), which are in contrast to the non-realistic assumption of independent bit values within signatures. Correlation is characterized by the aforementioned fraction, which is denoted as correlation factor ($corF$).

The real data that we used are two WWW traces, namely: (a) the ClarkNet data set, that contains two week's worth of all HTTP requests to the ClarkNet WWW server, and (b) the NASA data set, that contains two month's worth of all HTTP requests to the NASA Kennedy Space Center.[1] From these traces, we extracted user sessions, where each one was represented by a signature. The number of signatures for the ClarkNet data set is about 75,000 and the distinct items are 7200. For the NASA data

[1] The ClarkNet and NASA data sets can be obtained from http://ita.ee.lbl.gov/html/traces.html.

set, the number of signatures is equal to 100,000 and the distinct items are 1800. Both data sets present the aforementioned characteristic of correlation.

The performance metrics we used were: i) the accessed tree nodes, and ii) the compression ratio. Regarding the former (i) we did not employ a buffering scheme, since we wanted to focus on the complexity of query execution in terms of the number of nodes invoked during a query. Nevertheless, it is expected that the use of buffering will have the equivalent impact on all methods (thus, the relative performance difference will be preserved).

4.2 Experimental Results

In the first experiment, we examine performance for synthetic data with respect to the size of query, which is given in terms of its $wF(q)$. For each data signature s, its length F is equal to 512 and its $wF(s)$ is 0.05. The correlation factor was set up equal to 0.5. For the experiment we used an S-tree with a maximum of 120 signatures per node and with a minimum of 40 (denoted as 120/40), whereas the number of signatures were 50,000. The results are depicted in Figure 4. More specifically, the left part of Figure 4 depicts the number of disk accesses (denoted as DA), whereas the right part of Figure 4 depicts the CPU overhead due to decompression (for COMP only). As shown, a clear reduction in the number of disk accesses is attained by COMP. Moreover, the cost of decompression is low enough to guarantee that the total cost of query execution for COMP is much smaller than that for ORIG. As $wF(q)$ increases, the number of disk accesses is reduced for both COMP and ORIG. This is as expected, because with increased $wF(q)$ less data signatures satisfy the query and, thus, less nodes are examined (analogous reasoning can be followed for the reduction of the decompression time for COMP).

The second experiment is similar to first, however using different values for certain parameters. The factor weight wF(s) is equal to 0.9 and the correlation factor is equal to 0.3. The values of other parameters remain the same as in the previous experiment. The left part of Figure 5 shows the number of disk accesses for the COMP and the ORIG and right part of Figure 5 shows the time of decompression for the COMP. As shown, in this case, again, the number of disk accesses of COMP is less than that of ORIG (the decompression time is also quite low). Comparing the two experiments, however, we observe that at the second experiment the number of disk accesses for both methods is significantly increased. Moreover, the number of disk accesses is not decreased with increasing query factor weight. This is as expected, due to the high 'heavy' data signatures that were used, which lead to the creation of data signatures with low selectivity. Only in the case where the query factor weight is very high, number of signatures that satisfies the query (and the number of disk accesses) is relatively decreased.

In the following experiment we examined a different type of synthetic data sets to model basket market data, i.e., customer transactions, which were produced using the generator described in [AS94]. The characteristics of these data are that their signatures are sparse (they have few aces with respect to the length of the signature) and that the correlation between them is high (i.e., a case analogous to the one examined in the first experiment). Because the queries were generated from the data (to follow their distribution), the performance is examined with respect to the absolute weight of a query signature $w(q)$ instead of the query weight factor $wF(q)$. Following the nota-

tion of [AS94], the used dataset was the T10.I6.D100K, thus the number of data signatures was 100,000, the average number of items per customer transaction was equal to 10, and the other parameters were the default ones used by the generator. The results are illustrated in Figure 6. The left part of Figure 6 illustrates that COMP significantly outperforms ORIG in terms of required number of disk accesses, whereas the decompression time (right part of Figure 6) is low enough. This result indicates that for such interesting real-world cases, the proposed scheme can attains significant performance improvements, due to the characteristics of the data.

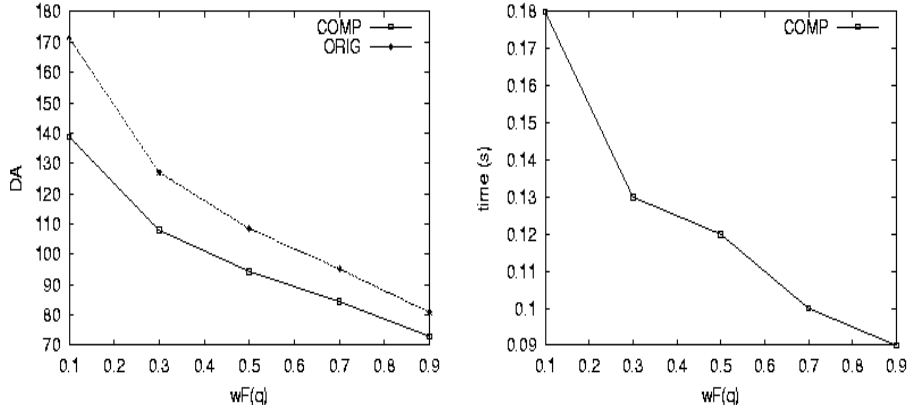

Fig. 4. Query performance with respect to query size for synthetic data. *Left*: Disk accesses w.r.t. $wF(q)$. *Right*: Decompression time

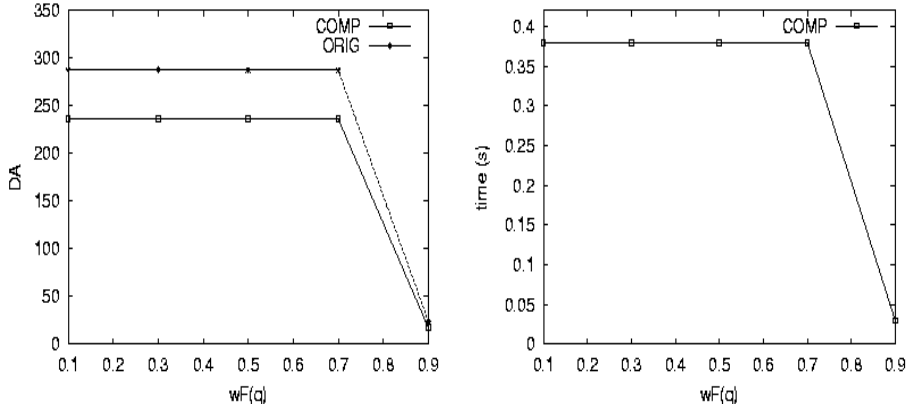

Fig. 5. Query performance with respect to query size for synthetic data. *Left*: Disk accesses w.r.t. $wF(q)$. *Right*: Decompression time

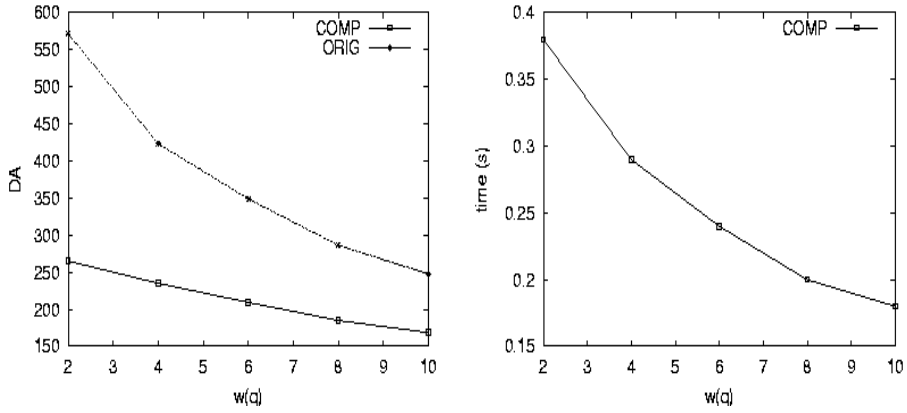

Fig. 6. Query performance with respect to query size for market basket data. *Left*: Disk accesses w.r.t. $w(q)$. *Right*: Decompression time

Also, we conducted two more experiments with real data sets, in order to verify the measurements obtained with synthetic data. Similar to the basket-market case, the queries were generated from the data signatures and their length is given with the $w(q)$ measure.

The results for the ClarkNet data set are illustrated in Figure 7. We observe that the disk accesses (left part of Figure 7) and the decompression time (right part of Figure 7) are similar with those in previous experiments. COMP clearly outperforms ORIG for all query sizes. The results for the NASA data set are depicted in Figure 8. As shown, for small queries ($w(q)$ less than 4) COMP is significantly better than ORIG. For higher weights, however, both methods require similar disk accesses, since the data signatures in this case are very sparse and the number of signatures that satisfy the query is very small. Therefore, compression may not pay-off for this cases, considering that, although small, the additional cost of decompression has to be paid.

Table 1. Compression ratios

Data set	**S-tree**	**Compressed S-tree**	**Compression Ratio**
Synth: wF(s)=0.05, corF=0.5	3155 KB	1798 KB	43%
Synth: wF(s)=0.9, corF=0.3	3162 KB	2964 KB	6.3%
Market basket data	5894 KB	1711 KB	71%
ClarkNet	6333 KB	794 KB	87.5%
NASA	4762 KB	911 KB	80.9%

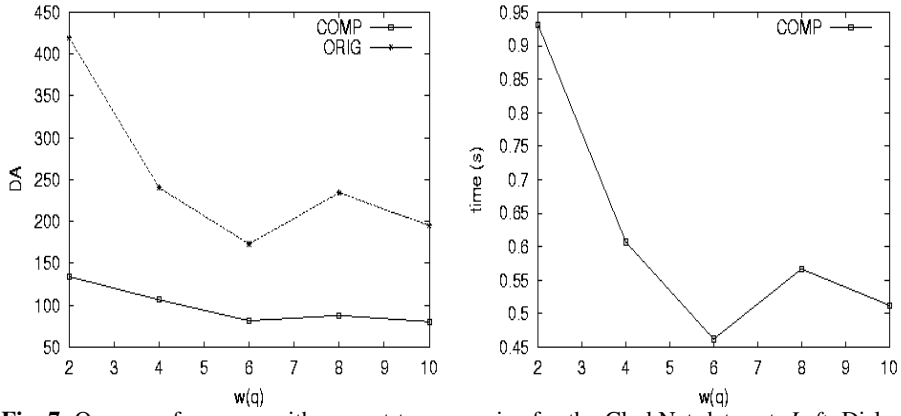

Fig. 7. Query performance with respect to query size for the ClarkNet data set. *Left*: Disk accesses w.r.t. $w(q)$. *Right*: Decompression time

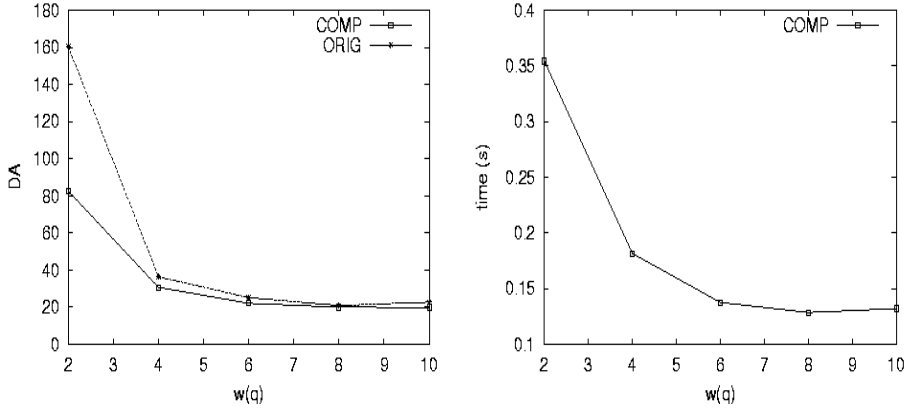

Fig. 8. Query performance with respect to query size for the NASA data set. *Left*: Disk accesses w.r.t. $w(q)$. *Right*: Decompression time

5 Conclusions and Future Work

We have examined the problem of compressing large signature tree structures aiming to significantly improvement in performance during query processing. We proposed ascheme that consists of the corresponding compression and decompression method. Both methods work on a per node basis and are based on the grouping of node entries according to their similarity, the production of sparse vectors, and the compression of the latter. The decompression method overcomes the problems of the straightforward approach (i.e., the decompression of the entire tree) and attains low overhead for this operation.

Detailed experimental results with real and synthetic data illustrated the gains reaped due to the proposed scheme. Especially for interesting real-world cases (basket market data, real traces), the proposed scheme presents significant improvements in terms of I/O cost, and very low CPU time overhead for decompression.

Future work includes the examination of more complicated techniques for the compression of sparse vectors [BK91]. Also, we will focus on the development of compression schemes that will support the construction of dynamic S-trees. The naive technique is the decompression of entire tree, the insertion of a signature and the compression again of entire tree. This technique is not effective. An efficient algorithm, similar with the one that we follow in the stage of decompression, is: decompress only the nodes that are invoked in the insertion and are determined from the insertion algorithm, which is executed simultaneously. More gain can be achieved by the observation that if the father of a node is the same before and after the insertion then all the nodes above this node (i.e., the nodes more close to the root) will be the same and therefore it is not need to compress them again. Moreover a great overhead is introduced in the stage of node split and therefore the criteria of signature insertion should be selected in order to minimize the number of splits. Further issues should be researched.

References

[AS94] R. Agrawal, R. Srikant. "Fast Algorithms for Mining Association Rules in Large Databases". *Proc. Conf. On Very Large Databases (VLDB'94)*, pp. 3–14, 1995.

[BBJK+00] S. Berchtold, C. Bohm, H. Jagadish, H.-P. Kriegel, J. Sander. "Independent Quantization: an Index Compression Technique for High Dimensional Data Spaces". *Proc. Conf. on Data Engineering (ICDE'2000)*, pp. 577–588, 2000.

[BK91] A. Bookstein, S. Klein. "Compression of Correlated Bit-Vectors". *Information Systems*, Vol.16, No.4, pp.387–400, 1991.

[CF84] S. Christodoulakis, C. Faloutsos. "Signature Files: An Access Method for Documents and its Analytical Performance Evaluation". *ACM Transactions on Office Information Systems*, Vol. 2, pp. 267–288, 1984.

[Depp86] U. Deppisch. "S-tree: A Dynamic Balanced Signature Index for Office Retrieval". *Proc. ACM SIGIR Conf.*, pp. 77–87, 1986.

[GRS98] J. Goldstein, R. Ramakrishnan, U. Shaft. "Compressing Relations and Indexes". *Proc. Conf. on Data Engineering*, pp. 370–379, 1998.

[MNT03] Y. Manolopoulos Y., A. Nanopoulos, E. Tousidou. "*Advanced Signature Indexing for Multimedia and Web Applications*", The Kluwer International Series on Advances in Databases Systems, Kluwer Academic Publishers, 2003, in print.

[NM02] A. Nanopoulos, Y. Manolopoulos. "Efficient Similarity Search for Market Basket Data". *The VLDB Journal*, Vol. 11, No. 2, pp. 138–152, 2002.

[NTVM02] M. Nascimento, E. Tousidou, C. Vishal, Y. Manolopoulos. "Image Indexing and Retrieval Using Signature Trees". *Data and Knowledge Engineering*, Vol. 43, No. 1, pp. 57–77, 2002.

[S-DR87] R. Sacks-Davis, K. Ramamohanarao. "Multikey Access Methods Based On Superimposed Coding Techniques". *ACM Transactions on Database Systems*, Vol. 12, No. 4, pp. 655–696, 1987.

[Teuh01] J. Teuhola. "A General Approach to Compression of Hierarchical Indexes". *Proc. Database and Expert Systems Applications (DEXA'2001)*, pp. 775–784, 2001.

[TBM02] E. Tousidou, P. Bozanis, Y. Manolopoulos. "Signature-based Structures for Objects with Set-valued Attributes". *Information Systems*, Vol. 27, No. 2, pp. 93–121, 2002.

[TNM00] E. Tousidou, A. Nanopoulos, Y. Manolopoulos. "Improved Methods for Signature-Tree Construction". *The Computer Journal*, Vol. 43, No. 4, pp. 301–314, 2000.

[WMB99] I. Witten, A. Moffat, T. Bell. "*Managing Gigabytes – Compressing and Indexing Documents and Images*". Morgan Kaufmann, 1999.

[Zezu88] P. Zezula. "Linear Hashing For Signatures Files". *Proc. IFIP TC6 and TC8 Symp. on Network Information Processing Systems*, pp. 192–196, 1988.