Performance Evaluation of Parallel S-trees

Eleni Tousidou

eleni@delab.csd.auth.gr

Michael Vassilakopoulos

mvass@computer.org

&

Yannis Manolopoulos

manolopo@delab.csd.auth.gr

Department of Informatics

Aristotle University

Thessaloniki, 540 06 Greece

Tel: +30 31 996363 Fax: +30 31 998419

Abstract

The S-tree is a dynamic height-balanced tree similar in structure to B^+trees. S-trees store fixed length bit-strings, which are called signatures. Signatures are used for indexing textbases, relational, object oriented and extensible databases as well as in data mining. In this article, methods of designing multi-disk B-trees are adapted to S-trees and new methods of parallelizing S-trees are developed. The resulting structures aim at achieving performance gain by accessing two or more disks simultaneously. In addition, two different searching techniques that exploit parallel disk accessing are devised. Performance results of experiments based on the new structures and searching techniques are also presented and commented.

Introduction

Signatures are bit strings of fixed length, produced by applying hash functions to objects values, e.g. data fields/records, text, images, voice, time-series etc. Basically, each object has a specified number of bits set to 1 in its signature. Due to hashing, it is probable that two different objects produce the same signature. Nevertheless, the signature stands for the original object as an abstract and fuzzy representation, and can be used for efficient processing of exact, as well as partial-match queries.

Modern information retrieval implementations utilize one, or more, of the following techniques: full text scanning, inversion, and the signature file (Faloutsos, 1985). Full text scanning introduces zero space overhead, but involves long response times. In the cases of inverted files and signature files, an intermediary index structure is utilized to provide direct links to relevant data. Recently, it has been reported that inverted files can be used to evaluate typical queries in less time that can signature files, and require less space and provide greater functionality as well (Zobel et al., 1999).

Apart from using signatures for text indexing, there are many practical database applications where signatures may be used. For example, representing an object with a signature is a technique that has been proposed for use in office filing (Christodoulakis et al., 1986; Tsichritzis et al., 1983) and hypertext systems (Faloutsos et al., 1990), relational, object-oriented databases and extensible databases (Chang & Schek, 1989; Ishikawa et al., 1993; Sacks-Davis et al., 1995; Yong et al., 1994), as well as in data mining (Andre-Jönsson & Badal, 1997).

In this paper we assume that we have a dynamic environment, where objects are inserted, deleted and updated. These objects may be any kind of conventional or multimedia data, as described above, and may vary in length and in structure. For example, text documents are sets of unformatted data elements, i.e. variable length words. Since our data set is dynamic, the set of signatures should be organized in a dynamic structure that guarantees high search performance.

The S-tree (Deppisch, 1986) is a height balanced dynamic structure, similar to a B^+tree (Comer, 1979), that has been proposed for storing signatures. The signatures of the objects are stored in the leaves of the tree. Each leaf stores similar signatures. For example, signatures that have many ones placed in common positions. These signatures are superimposed and OR-ed to produce the leaf signature. This signature is stored in the parent node of the leaf, as a key leading to this leaf. The process of using superimposition to produce a node signature as a key leading to the specific node applies to all the tree nodes. Each S-tree node, either internal or leaf, is a disk page. To keep performance high, the S-tree is a height balanced tree, that is all leaves are on the same level, and obeys certain limits on the minimum number of signatures a node may contain analogous to the limits of the B^+tree.

The S-tree has been proposed as a method for improving the performance of searches on a signature organized database. The goal of this article is to improve the performance of the S-trees even further by making use of a parallel disk environment. Thus, by storing the S-tree leaves in multiple disks, high I/O parallelism can be exploited. In (Seeger & Larson, 1991) the implementation of B-trees with multiple disks is examined. Three approaches are presented: *record distribution*, *large pages* and *page distribution*. The third approach, which is further elaborated to three variants, is shown to excel. In this article we adapt this approach to S-trees and elaborate it to five variants. Moreover, we present two methods that exploit the I/O parallelism of S-trees during searches. Lastly, we provide simulation results of the use of Parallel S-trees, in order to examine the I/O performance gain in comparison to non-parallel S-trees as well as to compare the I/O performance of the different proposed variants.

The organization of the rest of this article is as follows. In the next section, we present the S-tree in more detail. The third section refers to the new structure and its implementation with multiple disks using five variants of page distribution. In addition, two different methods of taking advantage of the parallelism during searches are presented. In the fourth section, experiments performed on the parallel S-tree are presented and discussed. Finally, the last section summarizes the results of our work and suggests future research trends.

S-trees

As has already been mentioned, each node of the S-tree corresponds to a disk page. Leaves appear on the same level and contain pairs of values. Each pair is made up of a signature, and a pointer or an identifier leading to an object. Each node, either leaf or internal, has a signature that is formed by superimposing and OR-ing the signatures it contains. Internal nodes contain pairs of values. Each pair corresponds to a child node of the specific internal node and is made up of the child node signature and a pointer to the child node.

S-tree nodes are characterized by two integers K and k. The root of the tree accommodates at least 2 and at most K pairs, unless it is a leaf. Every node, either internal or leaf, accommodates at least k and at most K pairs. Note that, unlike B-trees where $k{=}K/2$, it holds that $1 \leq k \leq K/2$. The height of an S-tree with n objects is at most

$\lceil \log_k n - 1 \rceil$. Note that, due to producing signatures of objects by hashing and the signatures of nodes by superimposing, it is possible that the same signature appears two or more times in the tree. Also, note that there is no ordering of entries within a node. In Figure 1, an example of an S-tree with height 3 is depicted. The signatures in the leaves are the ones created from the indexed objects and all have a constant number of ones. "Signature weight" represents the number of ones in a signature. Often, the weight of a signature s is symbolized by $\gamma(s)$. In this example, the signature weight is three at the leaf level, but varies from 3 to 6 in upper levels. In the sequel, we restrict the meaning of the notion of signature weight to the weight of signatures that appear in the leaves only.

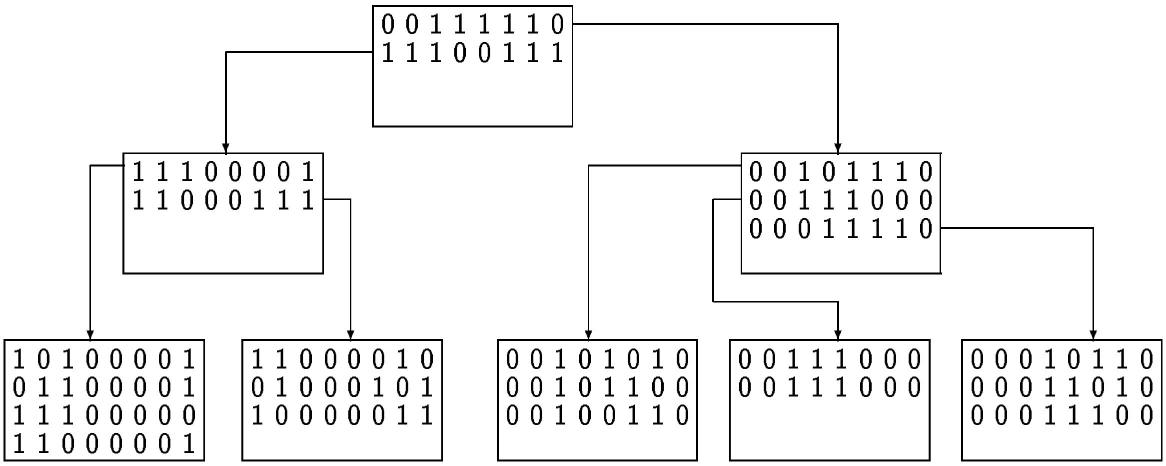

Figure 1: An example of an S-tree (K=4 and k=2).

Successful searches in S-trees 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 all the signatures of the root that contain ones in at least the same positions as the query signature, we follow the pointers to the children of the root. Note that more than one signature may satisfy this comparison. The process is repeated recursively for all these children down to the leaf level by opening multiple paths. Thus, at the leaf level, all the signatures satisfying the user query lead to the objects that may be the desired ones. In the case of an unsuccessful search, searching may stop early at some level above the leaf level if the signature of the user query has ones at positions where the stored signatures have zeros.

Insertions into S-trees proceed as follows. Given a signature to be inserted, we descend the tree downwards following a single path to a leaf where insertion will take place. At each step of our descent we choose the next node to be the one that has the smallest increase of ones in its signature, when it is superimposed with the signature under insertion. This strategy aims at minimizing the number of different paths that will "open" at each internal node during future searches. The new signature is inserted at the leaf reached at the end of the descent. If the insertion changes the signature of the leaf, this change is propagated to its ancestors. If before insertion the leaf is already full, it will have to be split in two. We want the resulting leaves to have low weight signatures that differ significantly. A metric of dissimilarity of two signatures $s1$ and $s2$ is their hamming distance $\delta(s1, s2) = \gamma(s1 \vee s2) - \gamma(s1 \wedge s2)$.

A heuristic is used that discovers the signature with the highest weight, called seed α, and the signature with the maximum number of ones in positions where seed α has zeros, called seed β. Seed α and seed β correspond to the two leaves that will result from the split. Each of the remaining signatures is superimposed with both seeds, the weight increases are calculated and each signature is stored in the node of the seed for which the weight increase is smaller.

Parallel S-trees

The performance of query processing in database systems is mainly determined by the performance of the secondary memory subsystem. This is apparently the case for searches in S-trees. We aim to increase the performance of searches by storing the nodes of an S-tree in multiple disks that are accessed in parallel.

We assume a system with a number of independent disks. There are two types of architecture for such a system. The disks are either connected by a common bus or by separate channels. The former suffers from the problem of bus contention, since only one disk can occupy the channel at a given time. This problem can be reduced significantly by providing enough local buffer space for each disk and a fast bus. Bus contention is difficult to model. For this reason, in the rest of the paper we assume either separate channels, or the use of local disk buffers that minimize delays due to bus contention.

In (Seeger & Larson, 1991) the implementation of B-trees with such a disk system is studied. Three approaches are presented:

- *record distribution*: a separate tree for a subset of the data element set is created for each disk,
- *large pages*: each tree node is partitioned and each partition is stored on a different disk, and
- *page distribution*: each tree node is stored on a different disk, following a certain distribution strategy.

Page distribution is the technique that we have adapted for S-trees. We present five different strategies of page distribution. Similar to the case of B-trees, the number of S-tree internal nodes is small in comparison to the total number of tree nodes (e.g. approximately 10%). Thus, we assume that during query processing internal nodes reside in main memory. Therefore, to evaluate these five strategies we count only accesses to the leaves that are stored in the multiple disks. In the sequel, we present the strategies of page distribution.

Random distribution. Each node is assigned randomly to one of the disks. The number of nodes on each disk is well balanced and the performance of the multi-disk system is never worse than a single disk equivalent, in case all the requested leaves during a search reside in one of the disks. However, there is no guarantee that during query processing the requested leaves will reside in different disks and that parallel disk accessing will be high.

Round Robin distribution. We follow a cyclic order when assigning nodes to disks. The node load among different disks is completely balanced. Nevertheless, the problem of unbalanced accesses to different disks remains equally severe in this strategy.

Neighbor page distribution. We call sibling leaves, the leaves that have the same internal node as parent. Due to the way it is decided in which leaf a new signature will be inserted and the way that nodes, either internal or leaves, are split when they overflow, sibling leaves are very likely to contain similar signatures. During a search query, it is very probable that more than one sibling leaf will be needed for answering the query. According to this strategy, we treat leaves in groups of siblings. First, we store each new node to a different disk, as long as there is a disk with no siblings. When all disks contain at least one sibling, we compute a distance function between the sibling not yet stored and each disk. In this article, we examine two functions giving the distance between a node and a disk:

a. the minimum hamming distance between the signature of this node and the signatures of its siblings already stored in the disk, or

b. the average of the hamming distances between the signature of this node and the signatures of its siblings already stored in the disk.

We store a leaf on the disk to which it has the maximum distance. For computing the distances between a node and each of the disks, we decided to use function (a), and when two or more disks have the same minimum distance, function (b). This strategy not only has a balanced node load, but a balanced access load among disks as well. However, after a number of insertions the probability that two leaves stored in the same disk will both be accessed during a search may increase. For example, Figure 2a illustrates an S-tree where the leaf L3 has been produced after a split from leaf L2. Let us use the names $s1$, $s2$ and $s3$ for the signatures of L1, L2 and L3, respectively. Suppose that there are two disks named D1 and D2 for storing leaves, that L1 has been stored in D1, that L2 has been stored in D2 and that a decision should be made about the disk in which L3

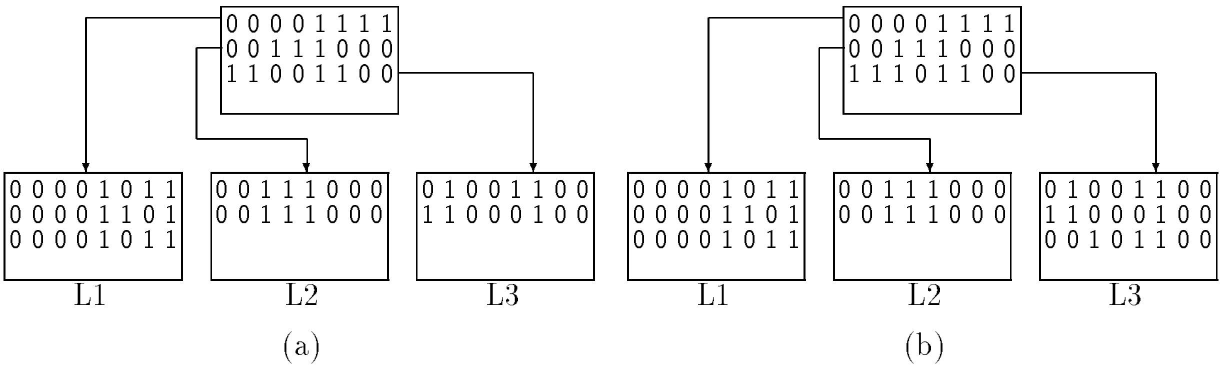

Figure 2: Example of a signature insertion that changes distances of nodes.

will be stored. Since $\delta(s3, s1)$=4 and $\delta(s3, s2)$=5, it follows that L3 will be stored in D2. In Figure 2b the insertion of the signature `01101100` in leaf L3 is depicted. However, this insertion changes signature $s3$. Now, it holds that $\delta(s3, s1)$=5 and $\delta(s3, s2)$=4. This means that L3 is stored in the same disk with L2, although it has lower probability to be accessed together with L1 during a search.

Local balancing distribution. If all the sibling leaves of an internal node have almost the same probability of being accessed, which is likely when the query signatures have low weight, we could simply balance the sibling leaves among different disks without taking into account the relationships among them. This is how the distribution of local balancing works. However, after many insertions, internal nodes may split and local balancing of sibling leaves is lost. For example, Figure 3a illustrates an internal node, which is a parent of leaves. Let us assume that the parameters of the specific tree are set to k=4 and K=10, and that the leaves are stored in 5 disks, D1 to D5. This node points to 10 leaves, L1 to

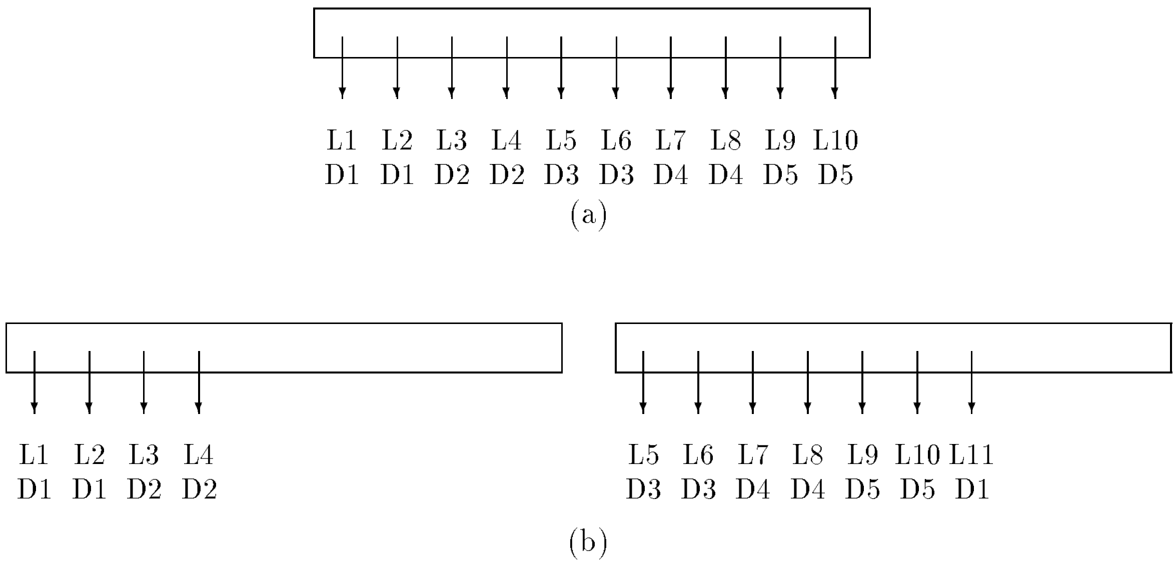

Figure 3: Example of a split of an internal node (parent of leaves).

L10. Below each leaf, the disk in which it is stored is mentioned. The balance of these leaves among the 5 disks is perfect: 2 leaves in each disk. Suppose that this internal node overflows, when an entry for leaf L11 is inserted. In Figure 3b a possible configuration of the two resulting internal nodes is depicted. The first one points to leaves L1 to L4 and the second one points to leaves L5 to L11. L11 will be stored in the first disk that stores none of its siblings, which is D1. Note that for both internal nodes, there is a very bad balance of sibling leaves among the 5 disks. Subsequent insertions, so that internal nodes will become almost full, will rebalance the numbers of sibling leaves among the disks.

Combination of neighbor page and local balancing distributions. Since neighbor page distribution efficiency is altered due to insertions, whereas local balancing distribution efficiency is altered due to splits, a combination of the two methods has been developed. This combination is closer to neighbor page distribution. When more than one disk are candidates according to the method of neighbor page distribution, we choose between them using local balancing and not the results of function (b). Another possibility for combining the two strategies and treating them equally, is to use neighbor page distribution of sibling leaves with few signatures and local balancing for sibling leaves that are almost full of signatures. The combination version that has been implemented is the former one.

The strategies of Random and Round Robin distributions have appeared in (Seeger & Larson, 1991) for B-trees. Also, a strategy called Local balancing distribution has appeared in (Seeger & Larson, 1991) for B-trees, however its adaptation to S-trees is totally different.

Searching techniques. Having parallelized the S-tree by one of the above methods, a searching technique that takes advantage of the multiple disks should be used. We devised two different techniques that are described in the following.

- Partially Parallel Search: During a search multiple paths to leaves are opened. The search algorithm assumes that we open a path to one level above the leaves, i.e. we reach a node that is a parent of leaves, and then perform parallel access of leaves in the multiple disk system. The opening and processing of the next path takes place only when processing of the current path is completed. This way, parallel access is achieved only among sibling leaves.

- Totally Parallel Search: A more sophisticated alternative is to open all paths and gather and keep in main memory all the identifiers of the leaves that should be accessed and then perform parallel access. Thus, parallel access among all leaves, either siblings or no siblings, is exploited under the extra cost of a limited amount of main memory.

Experimentation

The five strategies for parallelizing S-trees described above have been implemented in a Wintel computer using C++ and a large number of experiments have been conducted. We performed experiments for all combinations of the following quantities that express the use of Parallel S-trees in a real multi-disk system and a real world application.

- total number of signatures inserted: 50,000 and 100,000,

- signature size and signature weight measured in bits: 256-40, 512-120 and 1024-120,

- page size in bytes: 2048 and 4096,

- minimum page capacity, as percentage of the maximum page capacity: $k/K = 30\%$ and 40%,
- number of disks: 1, 5, 10, and 20,
- page distribution strategies: Random (RNDM), Round Robin (RROB), Local Balancing (LCBL), Neighbor page (NRPG) and Neighbor Page + Local Balancing (NPLB),
- searching technique: Partially Parallel (PP) or Totally Parallel (TP).

Each experiment had two steps. In the first step, signatures were randomly created, according to the restrictions of size and weight and were inserted in the tree. In the second step, for each tree a number of searches were conducted. The weight of the signatures searched, called the "Query Weight", ranged from 1 up to the weight of the signatures of the leaves. The signatures searched were created randomly and each time we performed 30 searches in each tree for each query weight. In all experiments we measured the performance in page accesses of both methods that were described in the previous section.

In our first set of experiments we measured the number of disk accesses versus the varying query weight of the signatures under both Partially Parallel and Totally Parallel searching. Note that usually each disk access transfers pages from more than one disk. In the next figures we show results for two representative numbers of disks, 10 and 20. The reason is that for less than 10 disks and for a node page size of 4K, the parallelism exploited is small, whereas for over 20 disks all methods seem to converge and the decrease in the number of disk accesses is insignificant.

The following graphs illustrate representative results derived during a large set of experiments conducted. As we can see in Figure 4, where we used 20 disks and counted the disk accesses according to the Partially Parallel technique, the Local Balancing method seems to outperform all other methods for a query weight up to 80. This can be explained by the fact that, due to its nature, LCBL method is the best one for the distribution of sibling leaves among disks. With a page size of 4K and 56 entries, splits are rather rare and LCBL method achieves the best performance. Note that, this is true because parallelism is exploited among sibling leaves only and the number of such leaves is always limited by K. For a query weight larger than 80, the random method (RNDM) achieves the best performance, while LCBL is very close to this optimum.

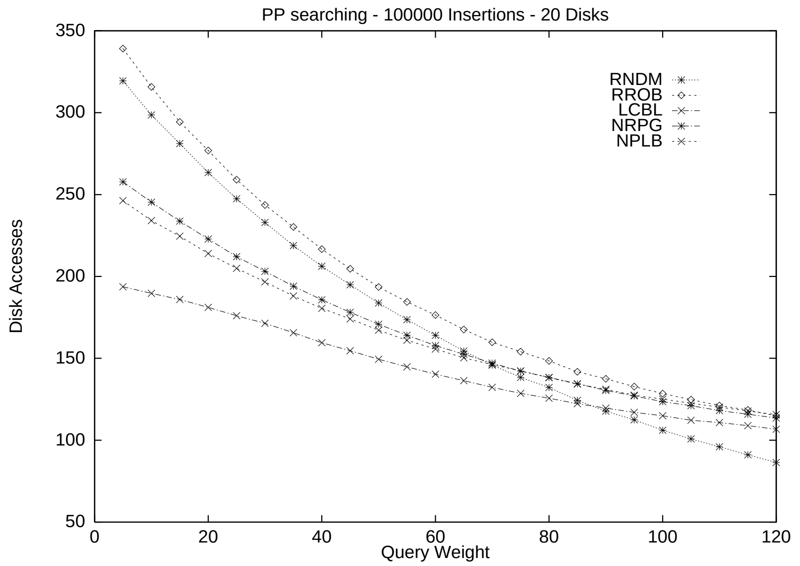

Figure 4: Number of accesses for all distribution methods, using PP Searching and 20 Disks, as a function of query weight. Number of signatures=100000, signature size=512, signature weight=120, k=17, K=56.

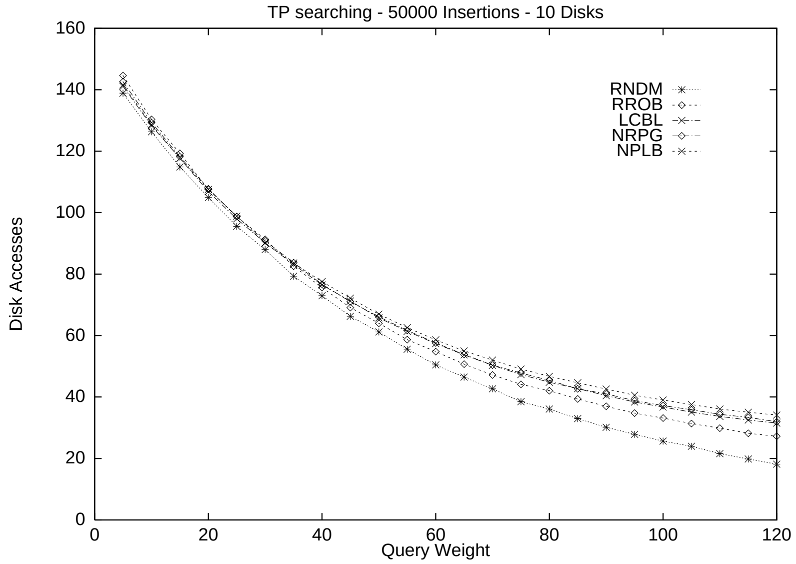

Figure 5: Number of accesses of all distribution methods, using TP Searching and 10 Disks, as a function of query weight. Number of signatures=50000, signature size=512, signature weight=120, k=17, K=56.

The results for the approach of Totally Parallel searching in Figure 5 show that the RNDM method's performance is also the best for a query weight larger than 60. For smaller query weights all methods appear to behave similarly. This is explained by noticing that for small query weights a large number of matches will arise, leading to a quite large number of leaves being visited. Due to the nature of the TP searching approach all methods are likely to balance its behavior.

In the second set of experiments we measured the number of disk accesses for an increasing query weight and for a varying number of available disks. The measurements were made under both searching approaches and for all methods. In Figures 6 and 7 we plotted the results only for the NPLB method, since the results for the rest of them are rather similar. For both searching techniques the first observation is that even for the case of only 5 disks, the gain is significant compared to the single disk approach with no

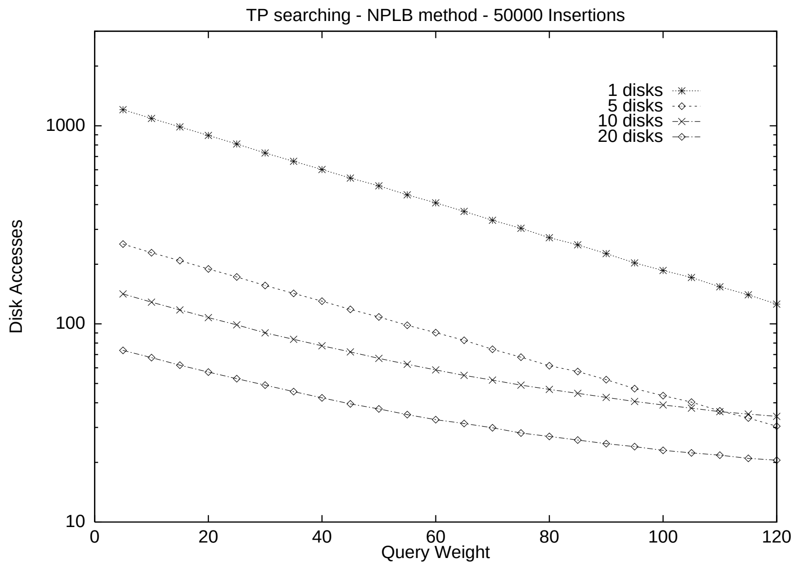

Figure 6: Number of accesses of NPLB method, using TP Searching, as a function of query weight. Number of signatures=50000, signature size=512, signature weight=120, k=17, K=56.

parallelism. Apart from this, we observe that in the PP searching method, for a query weight larger than the half of the signature weight, the difference in the decrease of the disk accesses among the different methods is quite insignificant. Similar results hold for the TP searching method, though we notice that for the case of 20 disks there is still a small decrease in the leaf accesses.

Conclusions

The S-tree structure can efficiently support exact and partial match queries in database systems. Storing the tree in a multi-disk system, where parallel disk accesses take place, increases the structure efficiency significantly. Five strategies for parallelizing the S-tree are presented in this article. Two searching techniques that exploit this parallelism are also presented. The experimentation shows that a limited number of disks, e.g. not more

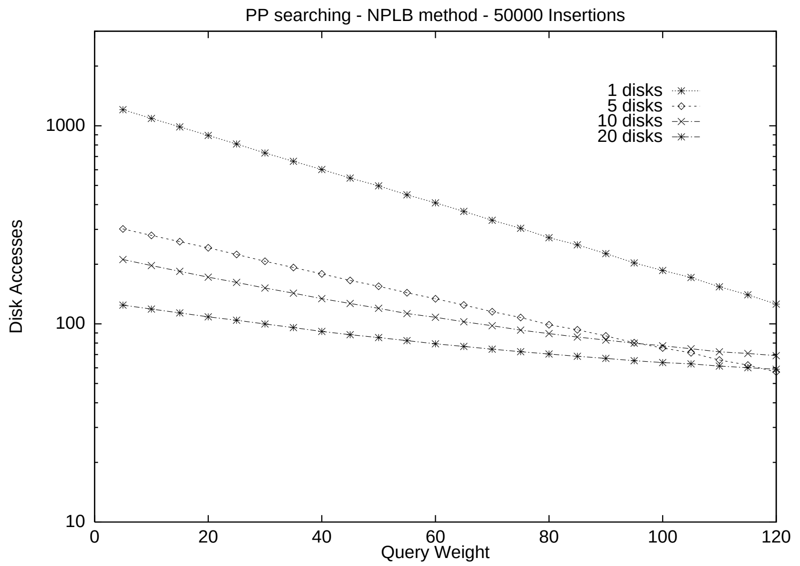

Figure 7: Number of accesses of NPLB method, using PP Searching, as a function of query weight. Number of signatures=100000, signature size=512, signature weight=120, k=17, K=56.

than 20, and a rather simple strategy, like Local Balancing or Random are sufficient for exploiting parallelism in disk accesses and increasing I/O efficiency.

Future research includes performing experiments that would model a shared bus architecture for the multi-disk system. In addition, enhanced distribution strategies that will adapt to insertions and splits could be designed and studied. Also, the problem of examining heuristics for page splits is an open problem. Moreover, comparison of parallel S-trees to frame-sliced partitioned parallel signature files (Ciaccia & Zezula, 1996; Grandi et al., 1992) is another interesting research direction.

Acknowledgments

This work has been partially supported by the bilateral Greek-Czech protocol. The authors would like to thank Dr. Pavel Zezula of the Technical University of Brno for his cooperation in the project on “Parallel textbases”. Thanks also due to Apostolos Papadopoulos for his helpful comments. The second author was supported by the State Scholarship-Foundation of Greece as a post-doctoral scholar of the latter foundation.

References

Andre-Jönsson H. & Badal D. (1997). “Using Signature Files for Querying Time-Series Data”. *Proceedings of the 1st European Symposium on Principles of Data Mining and Knowledge Discovery*.

Chang W.W. & Schek H.J. (1989). “A Signature Access Method for the STARBURST Database System”. *Proceedings of the 19th Very Large Data Base (VLDB) Conference*, 145-153.

Christodoulakis S., Theodoridou M., Ho F., Papa M. & Pathria A. (1986). “Multimedia Document Presentation, Information Extraction and Document Formation in MINOS - a Model and a System”. *ACM Transactions in Office Information Systems*, Vol.4, No.4, 345-383.

Ciaccia P. & Zezula P. (1996). “Declustering of Key-Based Partitioned Signature Files”. *ACM Transactions on Database Systems*, Vol.21, No.3, 295-338.

Comer D. (1979). “The Ubiquitous B-tree”. *ACM Computing Surveys*, Vol.11, No.2, 121-137.

Deppisch U. (1986). “S-tree: A Dynamic Balanced Signature Index for Office Retrieval”. *Proceedings of the ACM SIGIR Conference on Research and Development in Information Retrieval*, 77-87.

Faloutsos C. (1985). “Access Methods for Text”. *ACM Computing Surveys*, Vol.17, No.1, 49-74.

Faloutsos C., Lee R., Plaisant C. & Shneiderman B. (1990). "Incorporating String Search in a Hypertext System: User Interface and Signature File Design Issues". *Hyper-Media*, Vol.2, No.3, 183-200.

Grandi F., Tiberio P. & Zezula P. (1992). "Frame-Sliced Partitioned Parallel Signature Files". *Proceedings of the 1992 ACM SIGIR Conference*, 286-297.

Ishikawa Y., Kitigawa H. & Ohbo N. (1993). "Evaluation of Signature Files as Set Access Facilities in OODBs". *Proceedings of the 1993 ACM SIGMOD Conference*, 247-256.

Sacks-Davis R., Kent A., Ramamohanarao K., Thom J. & Zobel J. (1995). "Atlas: a Nested Relational Database System for Text Applications". *IEEE Transactions Knowledge and Data Engineering*, Vol.7, No.3, 454-470.

Seeger B. & Larson P.A. (1991). "Multi-Disk B-trees". *Proceedings of the ACM SIGMOD Conference*, 436-445.

Tsichritzis D., Christodoulakis S., Economopoulos P., Faloutsos C., Lee A., Lee D., Vanderbroek J. & Woo C. (1983). "A Multimedia Office Filing System". *Proceedings of the 9th Very Large Data Base (VLDB) Conference.*

Yong H.S., Lee S. & Kim H.J. (1994). "Applying Signatures for Forward Traversal Query Processing in Object-Oriented Databases". *Proceedings of the 10th IEEE International Conference on Data Engineering*, 518-525.

Zobel J., Moffat A.& Ramamohanarao K. (1999) "Inverted Files versus Signature Files for Text Indexing". *ACM Transactions on Databases Systems*, to appear.