Performance Evaluation of Parallel S-trees

Eleni Tousidou, Michael Vassilakopoulos & Yannis Manolopoulos
Aristotle University, Greece

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 discussed.

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 than can signature files, and require less space and provide greater functionality as well (Zobei 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-Jonsson & 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, i.e., 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 obey 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 \( \lceil \log_{2} n \rceil \). 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 3 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.

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 \( s_1 \) and \( s_2 \) is their hamming distance \( \delta(s, s_2) = \gamma(s \oplus s_2) - \gamma(s_1 \oplus s_2) \).

A heuristic is used that discovers the signature with the highest weight, called seed \( \alpha \), and the signature with the maximum number of ones in positions where seed \( \alpha \) has zeros, called seed \( \beta \). Seed \( \alpha \) and seed \( \beta \) 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