The SP-tree: A Clustered Index Structure for Efficient Sequential Access

Guang-Ho Cha[✉]

Department of Computer Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea
ghcha@snut.ac.kr

Abstract. We introduce the *SP-tree* that is a variant of a multidimensional index structure, with the object of offering efficient sequential disk access. The SP-tree is based on the index clustering technique called the *segment-page clustering* (*SP-clustering*). Most relevant index pages are widely scattered on a disk due to dynamic page allocation, and thus many random disk accesses are required during the query processing. The SP-clustering avoids the scattering by storing the relevant nodes contiguously in a *segment* that contains a sequence of contiguous disk pages and improves the query performance by offering sequential disk access within a segment. Experimental results demonstrate that the SP-clustering improves the query performance up to several times compared with the traditional ones with respect to the total elapsed time.

Keywords: Index clustering · Sequential disk access · Multidimensional index

1 Introduction

More than dozens of years of database research have resulted in a great variety of multidimensional indexing methods (MIMs). However, traditional MIMs tend to randomly access many index pages because the index pages are widely scattered on a disk due to dynamic page allocation. To avoid the performance degradation due to many random disk accesses, the related index nodes need to be clustered. However, existing MIMs do not take into account the clustering of *indices*. They take into consideration only the clustering of *data*. Moreover, the *dynamic* index clustering requires the on-line index reorganizations, and the overhead of the global index reorganization is excessive.

To overcome the drawbacks of the existing multidimensional indexing methods, we propose the *segment-page clustering* (*SP-clustering*) technique. The SP-clustering is based on the concept of *segments*. It considers the disk to be partitioned into a collection of segments. Each segment consists of a set of L contiguous pages on disk. A segment is the unit of clustering in the SP-clustering. All disk pages in a segment can be read by a single disk sweep, and thus it saves much disk startup and seek time. In the SP-clustering, all disk pages are addressed by a pair of (segment no, page no).

This research was supported by Basic Science Research Program through the National Research Foundation of Kore (NRF) funded by the Ministry of Education (NRF-2014R1A1A2059306).

This addressing scheme allows that pages as well as segments can be used as the disk access units. When random accesses are required or query ranges are very small, page-based disk accesses can be used instead of segment-based accesses.

2 Related Work

In the literature, some techniques for sequential disk access have been proposed. However, most of the work focuses on how to cluster data on a disk or how to physically maintain the data in sequence, and less attention has been paid to the clustering or sequential accessing for indexes.

The concept of the segment is similar to the idea of the *multi-page block* used in the SB-tree [5] and the bounded disorder (BD) access method [3, 4], which are variants of the B-trees, in the sense that they accommodate a set of contiguous pages and support multi-page disk accesses. However, this concept has not been applied to MIMs because it might consume the disk bandwidth excessively with increasing dimensionality. As an instance, let us suppose that a query range overlaps only a half on each dimension of the data region occupied by a segment. Then the wasteness of the disk bandwidth caused by reading a segment instead of reading individual pages is ½ (= 1 − ½) in one-dimensional case, while it is $1 - (½)^d$ in d-dimensional case. In fact, however, the multi-page disk reads such as segment reads are more needed in high dimensions because the probability that the query range overlaps with the regions covered by the index nodes increases with the dimensionality due to the sparsity of the domain space, and thus more disk pages are required to be read in higher dimensions. In addition, unlike the multi-page blocks used in the B-trees in which all index nodes as well as all data objects have total ordering among themselves, the index nodes within segments for MIMs have no linear order among them. This makes the design and maintenance, such as partitioning and merging, of the segments in MIMs more difficult than those of the multi-page block in the B-trees.

3 The SP-clustering

In this section we introduce the SP-clustering. To demonstrate the effectiveness of the SP-clustering, we apply it to the LSD-tree [1, 2] and call the resultant index tree the *SP-tree*. We should note that our focus is the SP-clustering not the SP-tree although we explain the SP-clustering through the SP-tree. The SP-clustering technique can be applied to most MIMs including the R-trees.

3.1 The Structure of the SP-tree

The SP-tree is a multidimensional index structure to index d-dimensional point data, and its underlying structure is the LSD-tree. We chose the LSD-tree for implementation of the SP-clustering because we were dealing with d-dimensional point data and the LSD-tree has high fanout. The SP-tree considers the disk to be partitioned as a collection of segments. Segments are classified into *nonleaf segments* and *leaf segments*. The nonleaf segment accommodates nonleaf nodes of the index tree and the leaf segment holds the leaf nodes. The reasons why we separate the segments into two kinds are twofold: it simplifies the design of the index structure and it encourages the upper part of the index structure to reside in the main memory when we cache the index into the main memory. We call the nonleaf segment *n-segment* and the leaf segment *l-segment*.

Each segment consists of a set of f (e.g., =32) contiguous pages on disk, that can be read or written with a single sweep of the disk arm. From the first page encountered by the disk head in reading and writing a segment to the last one in the segment, the pages are numbered 1, 2, …, f. A segment has the following properties:

- A segment consists of a set of f nodes which reside on contiguous pages on disk. The number f is called the *fanout* of a segment.
- k, $1 \leq k \leq f$, nodes falling in a segment are filled contiguously from the beginning of the segment.
- The SP-tree reads k nodes from a segment at a time rather than all f nodes, and thus it saves the disk bandwidth.
- Every node of the SP-tree sits on a segment.
- Leaf nodes reside in an *l*-segment and nonleaf nodes are in an *n*-segment.

Consider Fig. 1. A node at level j-1 (level 0 is the root) contains pointers (ptr 1, ptr 2,…, ptr k) to child nodes (pages) and the separators ((dim 1, pos 1), …, (dim k, pos k)). A separator contains a pair of a *split dimension* and a *split position* in the dimension. The entry M denotes the number of entries in the node. The contiguous sequence of page numbers, ptr 1 through ptr k, in the node at level $j - 1$ points to nodes at level j, node 1 through node k, which sit on the first k pages of a single segment, i.e., $M = k$. A specific node can be accessed directly by a pair of (segment number, page number), and k nodes can be accessed sequentially by the segment number and the value of M.

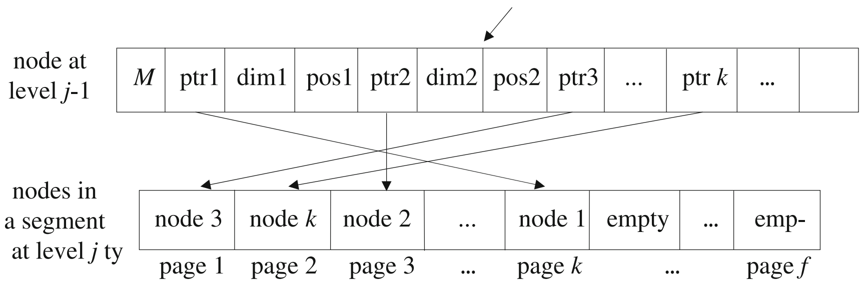

Fig. 1. Contiguous pages on a segment

3.2 Building the SP-Tree

The SP-tree has a hierarchical index structure. As usual for index structures which support spatial accesses for point data, the SP-tree divides the data space into pairwise disjoint cells. With every cell a data page is associated, which stores all objects contained in the cell. In this context, we call a cell a *directory region*.

Successive parts of Fig. 2 show how the SP-tree grows and how its nodes are clustered in a segment. When the first entry is inserted, a single page of an *l*-segment is allocated for the first node of the SP-tree. This node is a root node as well as a leaf node. The figures in the left side of Fig. 2 show the procedure of partitioning the 2-dimensional data space. We assumed that the range of each dimension is 0 to 100, and a pair of numbers on the directory regions indicates (*l*-segment number, page number).

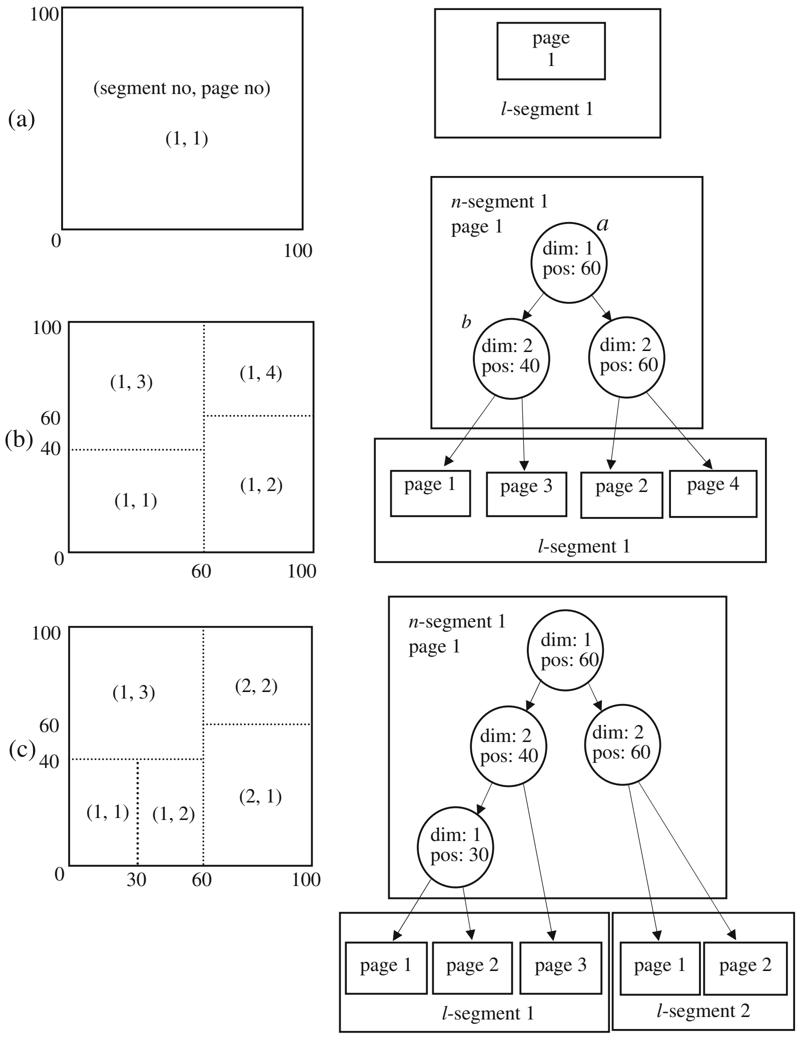

Fig. 2. The growth of the SP-tree

Successive entries are added to the node until an insert forces a split in the node. This node is then split into two leaf nodes which occupy page 1 and page 2 of the *l*-segment 1. An *n*-segment is allocated and the first page of the *n*-segment is assigned for new root node. The root node now contains a single separator and two pointers. A separator contains the information about the *split dimension* and the *split position* in the dimension. In the example of Fig. 2, the split is performed at position 60 in dimension 1. With subsequent insertions, overflows occur in the *l*-segment 1 and they cause the node split. Whenever a node split occurs, the SP-tree looks for an empty page on the segment containing the node receiving the insert. As shown in Fig. 1, this will be the page number $k + 1$ in the containing segment, where k nodes already exist. The SP-tree keeps the information in the node which tells us how many pages are occupied in each segment, i.e., M in Fig. 1. If an empty page exists, we place the new node created by the split on that page. If there is no empty page in the segment, then a *segment split* is necessary. A new segment S is allocated, and the overfull segment containing the splitting node is read into the memory. Then the $f + 1$ nodes of the segment are distributed into two segments.

3.3 Segment Split Strategy

An important part of the insertion algorithm of the SP-tree is the segment split strategy which determines the split dimension and the split position. First, the SP-tree finds the internal node u which (directly or indirectly) plays a role of root for the overfull segment R. The separator of the internal node u has the dimension and the position to split the segment R. In Fig. 2(b), for example, if a new entry is inserted into the page (1, 1) and it causes the *l*-segment 1 to overflow, the SP-tree finds the internal node which plays a role of root of the *l*-segment 1, it is the internal node a in the case of Fig. 2(b). Since the SP-tree maintains an array to save the traversal path from the root to the target page where a new entry to be inserted, it is not difficult to find the internal node that plays a role of root for the overfull segment. Starting from the root of the SP-tree, we check if the overfull segment can be split into two when we apply the separator of the current internal node to split the segment. If the segment can be split using the separator, the corresponding internal node is selected as the root node of the overfull segment, and the segment is split. The data pages belonging to the right children of u are reallocated to the front positions of a new segment S, and the remaining pages are moved forward so that they fall on the front pages of the segment R. As a result of this segment split strategy, the data pages under the same internal node are collected in the same segment.

Consider Fig. 3. It shows the effect of a nonleaf page split. In Fig. 3(a) we are to insert a new index entry into the nonleaf page P due to a page split in lower level segment S'. If the page P overflows, the SP-tree allocates a new page P' and partitions the entries in P all but the local root entry u including a new index entry into two pages P and P'. After the partition, the local root entry u that was in P is promoted to the parent page, and the two pages P and P' include the entries belonged to the left and the right subtrees of u, respectively (see Fig. 3(b)).

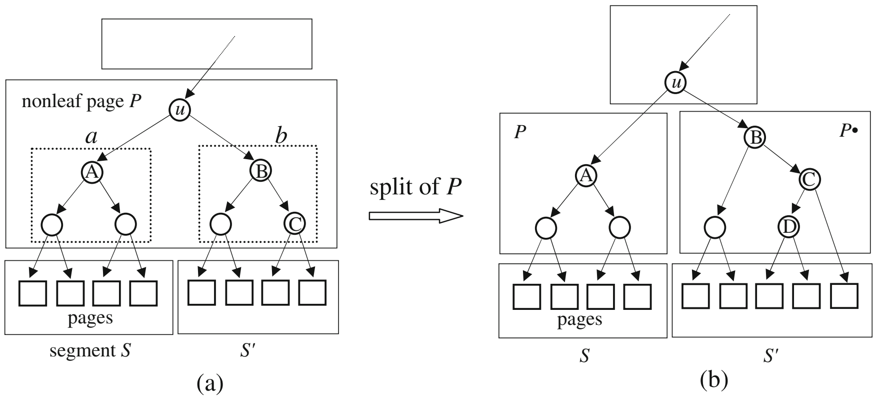

Fig. 3. Effect of a nonleaf page split

The SP-tree is a binary tree, and the entries in the left subtree and the right subtree of the local root entry in a certain nonleaf page point to the pages in different segments unless all the pages referenced lie in the same single segment. For example, the entries in the subtrees *a* and *b* in Fig. 3(a) point to the pages in different segments *S* and *S*′, respectively. When the page *P* is split, as shown in Fig. 3(b), the index entries that point to the pages in a certain segment are inserted into the same page. Thus, in the SP-tree, the references of pages that belong to a segment are stored in a single nonleaf page.

We should note that the SP-tree does not cause a split of lower level segment that is not full due to a split of upper nonleaf page. In the SP-tree, the fanout of a nonleaf page (it is often several hundreds) is far larger than that of a segment (it is usually several tens). In other words, there are generally many segments under a nonleaf index page. Therefore the segments referenced by the left and the right parts of the control node (i.e., local root) of the upper nonleaf page that is full are already different as shown in Fig. 3(a). Thus, after the upper level page split, it is not the case to split lower level segments to preserve the property that the references of pages that belong to a segment are stored in the same nonleaf page.

4 Performance Evaluation

To demonstrate the practical effectiveness of the SP-clustering, we implemented the SP-tree by using 64-KB segments. We performed an extensive experimental evaluation of the SP-tree and compared it to the pure LSD^h-tree. The experimental results demonstrate that in most cases the SP-clustering is superior to traditional MIMs. For random queries such as exact-match queries and nearest neighbor queries, there is little performance difference between the SP-tree and the LSD^h-tree.

5 Conclusions

We have introduced the SP-tree for range queries. The performance advantage of the SP-clustering comes from saving much disk startup time. Moreover, storing a sequence of index pages contiguously within a segment provides a compromise between optimal index node clustering and the excessive full index reorganization overhead. Thus, the SP-clustering methods may be used as an alternative index clustering scheme. The SP-clustering is so generic that it may be applied to most MIMs.

References

1. Henrich, A.: The LSDh-tree: an access structure for feature vectors. In: Proceedings of the International Conference on Data Engineering, pp. 362–369 (1998)
2. Henrich, A., Six, H.-W., Widmayer, P.: The LSD-tree: spatial access to multidimensional point and non-point objects. In: Proceedings of the ICDE, pp. 44–53 (1989)
3. Litwin, W., Lomet, D.B.: The bounded disorder access method. In: Proceedings of the IEEE International Conference on Data Engineering, pp. 38–48 (1986)
4. Lomet, D.B.: A simple bounded disorder file organization with good performance. ACM Trans. Database Syst. **13**(4), 525–551 (1988)
5. O'Neil, P.E.: The SB-tree: an index-sequential structure for high-performance sequential access. Acta Informatica **29**, 241–265 (1992)