

Towards Optimal Indexing for Segment Databases

E. Bertino B. Catania

Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico 39/41
20135 Milano, Italy
e-mail: {bertino,catania}@dsi.unimi.it

B. Shidlovsky

Rank Xerox Research Center
Grenoble Laboratory
6, chemin de Maupertuis
38240 Meylan France
e-mail: chidlovskii@grenoble.rxrc.xerox.com

Abstract

Segment databases store N non-crossing but possibly touching segments in secondary storage. Efficient data structures have been defined to determine all segments intersecting a vertical line (stabbing queries). In this paper, we consider a more general type of query for segment databases, determining intersections with respect to a generalized segment (a line, a ray, a segment) with a fixed angular coefficient. We propose two solutions to solve this problem. The first solution has optimal $O(\frac{N}{B})$ space complexity, where N is the database size and B is the page size, but the query time is far from optimal. The second solution requires $O(\frac{N}{B} \log_2 B)$ space, the query time is $O(\log_B \frac{N}{B} (\log_B \frac{N}{B} + \log_2 B + IL^*(B)) + \frac{T}{B})$ time, which is very close to the optimal, and insertion amortized time is $O(\log_B \frac{N}{B} + \log_2 B + \frac{1}{B} \log_B^2 \frac{N}{B})$, where T is the output size of the query, and $IL^*(B)$ is a small constant, representing the number of times we must repeatedly apply the \log^* function to B before the result becomes ≤ 2 .

1 Introduction

Background. Advanced declarative programming features and new application domains, dealing with spatial, geographical, and temporal data, require the development of new data structures for querying and updating such data, with time and space comparable to those of data structures for relational databases. Complexity of the various operations is expressed in terms of *input-output (I/O) operations*. An I/O operation is the operation of reading/writing one block of data from or to a disk. Other parameters are respectively: N , the number of items in the database; B , the number of items per disk block; T , the number of items in the problem solution; $n = N/B$, the optimal number of blocks required to store the database; $t = T/B$, the optimal number of blocks to access for reporting the problem result. For 1-dimensional data, efficient data structures like B-trees and B⁺-trees [7] process range queries in $O(\log_B n + t)$ I/O operations, require $O(n)$ blocks of secondary storage, and perform insertions/deletions in $O(\log_B n)$ I/O operations. All these complexities are worst-case.

A large amount of work has been carried out to obtain optimal I/O complexities for the so called “point databases”. For such databases, which typically contain N points on the plane, several external data structures have been developed. Those data structures are characterized by an I/O complexity for search and update operations comparable to the internal memory results [9, 12, 17, 19].

Compared to the “point database” case, much less work has been carried out for the so called “segment databases” which represent a more general case. A segment database stores in

secondary storage N non-crossing but possibly touching plane segments (for brevity, called NCT segments). Segment databases are the basis for data representation in several large scale applications, including spatial databases and geographical information systems (GIS) [16], temporal databases [13] and constraint databases [11]. Among all possible applications, GIS certainly represent the main target of segment databases. Indeed, GIS databases often store data as layers of maps, where each map is typically stored as a collection of NCT segments.

Some relevant query types for segment databases can be reduced to a *stabbing query* on a set of 1-dimensional intervals. Given a set of input intervals and a point q , a stabbing query for q retrieves all segments that intersect q . As an example of query that can be reduced to a stabbing query, consider the query that, given a vertical line l , retrieves all segments intersected by l . This query can be reduced to a stabbing query against 1-dimensional segments which are x -projections of the database segments. Efficient solutions to the problem of external stabbing queries have been discussed in [12, 18]. Recently, the optimal solution to the problem has been proposed in [3]. Optimal external memory algorithms for some other segment problems have been presented in [2].

The problem. A more general and relevant problem in segment databases (especially for GIS) is to determine all segments intersecting a given query segment. In this paper we go one step towards the solution of this problem by solving a weaker problem, consisting in determining all segments intersected by a given generalized query segment (a line, a ray, a segment), having a fixed angular coefficient. Without leading the generality of the discussion, we assume that query segments are vertical¹. The corresponding query is called *VS query* (see Figure 1).

There exists a difference in the optimal time complexity between VS and stabbing queries even in internal memory. A space optimal solution for solving VS queries in internal memory has $O(\log^2 N + T)$ query complexity, uses $O(N)$ space and performs updates in $O(\log N)$ time [5] (with update we mean the insertion/deletion of a segment non-crossing, but possibly touching, the already stored ones). On the other hand, the optimal solution for solving stabbing queries in internal memory has $O(\log N + T)$ query complexity, uses $O(N)$ space and performs updates in $O(\log N)$ time [6]. VS queries are therefore inherently more expensive than stabbing queries. In this paper we propose a $O(n \log_2 B)$ space solution having query I/O complexity very close to $O(\log_B^2 n + t)$. The solution is proposed for static and semi-dynamic (thus, allowing insertion) cases.

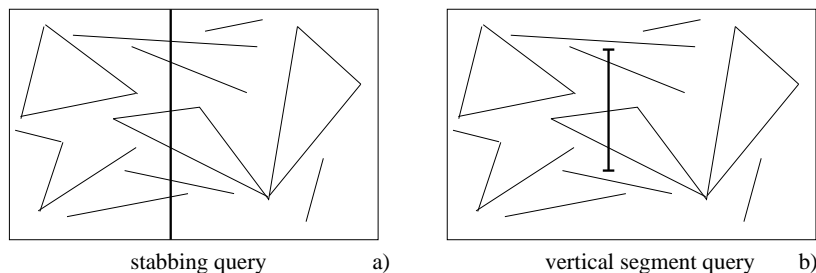


Figure 1: Vertical line queries vs vertical segment queries.

The proposed results. The data structures we propose to solve VS queries are organized according to two levels. At the top level, we use a primary data structure (called *first-level data structure*). One or more auxiliary data structures (called *second-level data structures*) are associated with each node of the first-level data structure. The second-level data structures are tailored to efficiently execute queries on a special type of segments, called *line-based segments*.

¹If the query segment is not vertical, coordinate axes can be appropriately rotated.

A set of segments is line-based if all segments have an endpoint lying on a given line and all segments are positioned in the same half-plane with respect to such line. Thus, the main contributions of the paper can be summarized as follows:

1. We propose a data structure to store and query line-based segments, based on *priority search trees* (PST for short) [6, 9], similar to the internal memory data structure proposed in [5].

In internal memory, priority search trees [14] are used to answer 3-sided queries (a special case of the 2-dimensional range searching problem) in optimal space, query, and update time. Given a set of points in the plane and a rectangular region open in one of its sides, the corresponding 3-sided query returns all points contained in the (open) rectangle. All proposals to extend priority search trees, for use in secondary storage, do not provide both query and space optimal complexities [9, 17, 19]. In [9], a solution with $O(n)$ storage and $O(\log n + t)$ query time was developed. Two techniques have been defined to improve these results. *Path-caching* allows to perform 3-sided queries in $O(\log_B n)$, with $O(n \log_2 B \log_2 \log_2 B)$ space. A space optimal solution to implement secondary storage priority trees is based on the *P-range tree* [19] and uses $O(n)$ blocks, performing 3-sided queries in $O(\log_B n + IL^*(B) + t)$ and updates in $O(\log_B n + \frac{\log_B^2 n}{B})$, where $IL^*(B)$ is a small constant, representing the number of times we must repeatedly apply the \log^* function to B before the result becomes ≤ 2 ². We use a P-range tree to reduce the time and space complexity of the structure we propose to store line-based segments.

2. We propose two approaches to the problem of VS queries. They can be characterized as follows:

- In the first solution, the first-level structure is a binary tree, whereas the second-level structure, associated with each node of the first-level structure, is a pair of priority search trees, storing line-based segments in secondary storage. This solution uses $O(n)$ blocks of secondary storage and answers queries in $O(\log n(\log_B n + IL^*(B)) + t)$ I/O's. We also show how updates on the proposed structures can be performed in $O(\log n + \frac{\log_B^2 n}{B})$ amortized time.
- To improve the query time complexity of the first solution, we replace the binary tree at the top level with a secondary storage interval tree [3]. The second-level structures are based on priority search trees for line-based segments and segment trees, enhanced with fractional cascading [4]. This solution uses $O(n \log_2 B)$ space and answers queries in $O(\log_B n(\log_B n + \log_2 B + IL^*(B)) + t)$ time. We also show how the proposed structure can be made semi-dynamic, performing insertions in $O(\log_B n + \log_2 B + \frac{\log_B^2 n}{B})$ amortized time.

Organization of the paper. The paper is organized as follows. Section 2 proposes a data structure for line-based segments. Section 3 introduces the first solution to the problem of indexing VS queries. This solution is then extended in Section 4. Finally, Section 5 proposes some conclusions and outlines open problems. Two appendices are also inserted in the paper: Appendix A presents some algorithms whereas Appendix B presents some additional figures.

²Unless otherwise stated all logarithms are given with respect to base 2.

2 Data structures for line-based segments

Let S be a set of segments. S is *line-based* if there exists a line l (called *base line*) such that each segment in S has (at least) one endpoint on l and all the segments in S having only one end-point on l are located on the same half-plane with respect to l .

In the following we construct a data structure for storing line-based segments in secondary storage and retrieving all segments intersected by a query segment q which is parallel to the base line.

A query q may be a segment, a ray, or a line. A line query is defined by its distance from the base line³. Ray and segment queries are given by a line, which is parallel to the base line, and one or two points on it, representing respectively the ray or the segment endpoints. Since a segment query represents the most complex case, in the following we focus only on such type of queries. Moreover, without loss of generality, through all Section 2, we restrict the presentation to horizontal base lines. This choice simplifies the description of our data structure making it coherent with the traditional way of drawing data structures. The query thus becomes a horizontal segment as well.

The solution we propose for storing a set of line-based segments is based on the fact that there exists an obvious relationship between a 3-sided query against a set of points and a segment query against a set of line-based segments on the plane (see Figure 2). A segment query defines in a unique way a 3-sided query on the point database corresponding to all segment endpoints not belonging to the base line. On the other hand, the bottom segment of a 3-sided query on such point database corresponds to a segment query on the segment database. However, these corresponding queries do not necessarily return the same answers. Indeed, although both queries often retrieve the same data (segment 1 in Figure 2), this is not always true. The intersection of a segment with the query segment q does not imply that the segment endpoint is contained in the 3-sided region (segment 2 in Figure 2). Also, the presence of a segment endpoint in the 3-sided region does not imply that the query segment q intersects the segment (segment 3 in Figure 2). Despite these differences, we show that solutions developed for 3-sided queries (path-caching and P-range trees [17, 19]), based on priority search trees, can be successfully applied to line-based segments as well.

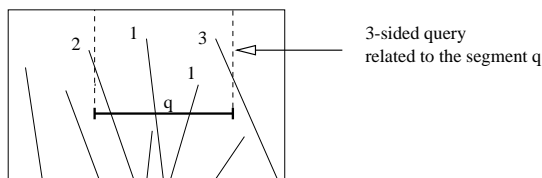


Figure 2: A segment query on a set of line-based segments vs a 3-sided query on the endpoint set of the same segments.

As in the approach presented in [17, 19] for point databases, in order to define priority search trees for a set of line-based segments, a binary decomposition is first used and algorithms for retrieving all segments intersected by the query segment are developed. As a result, we obtain a binary tree structure in secondary storage of height $O(\log n)$, which meets all conditions required in [17, 19] for applying any advanced technique between path-caching and P-range tree.

Data structure. Let S be a set of N line-based segments. We first select a number B of

³We assume that the query segment belongs to the same half-plane where segment endpoints belong. Otherwise, no segment intersects the query.

segments from S with the topmost y -value endpoints and store them in the root r of the tree Tr under construction, ordered with respect to their intersections with the base line, together with a separator low , which is a horizontal line separating the selected segments from the others⁴. The set containing all other segments is partitioned into two subsets containing an equal number of elements. The top segment in each subset is then copied into the root. These segments are denoted respectively by $left$ and $right$. Line low for a generic node v is denoted by $v.low$. A similar notation is used for $left$ and $right$ (see Figure 3).

The decomposition process is recursively repeated for each of the two subsets. Like external priority search tree in point databases, the resulting tree Tr is a balanced binary tree of height $O(\log n)$, occupying $O(n)$ blocks in external storage. The difference, however, is that no subtree in Tr defines a rectangular region in the plane. Moreover, in a point database, a vertical line is used as separator between points stored in left and right subtrees. Instead, in a segment database, the line which separates segments stored in left and right subtrees is often biased (see Figure 3).

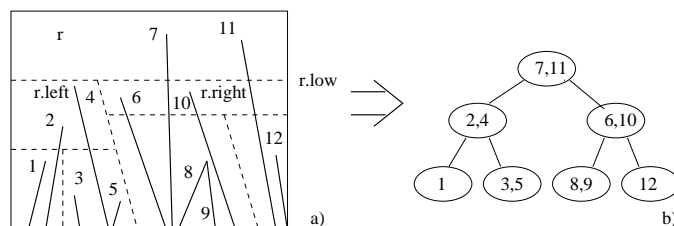


Figure 3: External PST for line-based segments (a) and corresponding binary tree (b), assuming $B = 2$. Non-horizontal dashed lines do not exist in the real data structure and are given only for convenience. Moreover segments $v.left$ and $v.right$, as well as line $v.low$, are shown only for the root r .

Search algorithm. Let q be a horizontal query segment. We want to find all segments intersecting q . The search algorithm is based on the comparison of q with stored segments⁵. The search is based on two functions $Find$ and $Report$. Function $Find$ is to locate the deepest-leftmost (deepest-rightmost) segment intersected by the query q , with respect to its storage position in Tr , and the node in Tr where the segment is located. Function $Report$ then uses the result of function $Find$ to retrieve all segments in Tr intersected by q , starting from the deepest-leftmost and deepest-rightmost segments intersecting the query. The algorithms for such functions are presented in Appendix A. They are similar to the ones presented in [5] for the internal memory intersection problem, and satisfy the following properties:

- Function $Find$ maintains in a queue Q the references to the nodes that should be analyzed to find the deepest-leftmost (the deepest-rightmost) segment intersecting the query segment. It can be shown that Q refers at most two nodes on each level of Tr , thereby assuring that the answer is found in $O(\log n)$ steps. The space utilization is $O(1)$.
- Function $Report$ determines the deepest-leftmost and the deepest-rightmost segments intersecting the query, using function $Find$. Then, it visits the subtree rooted at the common ancestor of the nodes containing such segments. It can be proved that the number of nodes of such subtree, containing at least one segment non-intersecting q , is in $O(\log n + t)$.

⁴Note that the construction guarantees that each node is contained in exactly one block.

⁵Note that this is different from the approach usually used in PST for point databases. In that case, the comparison is performed against region boundaries. Such an approach is not possible for line-based segments, since no rectangular region is related to any subtree of Tr .

Lemma 1 *Let S be a set of line-based segments. Let q be a query segment. Let Tr be the priority search tree for S , constructed as above. Then:*

1. *Function $Find$ returns the deepest-leftmost (the deepest-rightmost) segment in S intersected by q and the node it belongs to in $O(\log n)$ I/O's.*
2. *Let bl_l and bl_r be the blocks in Tr containing the deepest-leftmost and deepest-rightmost segments intersected by q , respectively. All T' segments in S intersected by q can be found from Tr , bl_l and bl_r in $O(\log n + \frac{T'}{B})$ I/O's. \square*

The following result summarizes the costs of the proposed data structure.

Lemma 2 *N line-based segments can be stored in a secondary storage priority search tree having the following costs: (i) storage cost is $O(n)$; (ii) horizontal segment query cost is $O(\log n + t)$, where t is the number of the detected intersections. \square*

Despite the difference in the query results between a 3-sided query on a point database and a segment query on a segment database (see Figure 2), any of the path-caching or P-range tree methods can be applied for reducing the search time using an external PST in a segment database. Since in the next section we will use an external PST on each level of the first-level data structure we are going to develop, we choose a linear memory solution based on P-range trees [19], to obtain an optimal space complexity in the data structures for storing line-based segments.

The adaptation of the P-range tree technique to a segment database requires only one deviation from the technique described in [19]. It substitutes the vertical line separator with the queue Q and several procedures needed for the queue maintenance. Then, a comparison of a query point against a vertical line-separator in a point database is substituted with the check of at most two candidates in query Q during the search in a segment database. Since the detection of the next-level node and the queue maintenance in a segment database takes $O(1)$ time (see above) this substitution does not influence any properties of the P-range tree technique. This proves the following lemma.

Lemma 3 *N line-based segments can be stored in a secondary storage data structure having the following costs: (i) storage cost is $O(n)$; (ii) horizontal segment query cost is $O(\log_B n + IL^*(B) + t)$ I/O's; (iii) update amortized cost is $O(\log_B n + \frac{\log_B^2 n}{B})$. \square*

3 External storage of NCT segments

In order to determine all NCT segments intersecting a vertical segment, we propose two secondary storage solutions, based on two-level data structures (denoted by 2LDS). Second-level data structures are based on the organization for line-based segments presented in Section 2. In the following, we introduce the first proposed data structure (the second one will be presented in Section 4).

First-level data structure. The basic idea of the first solution is to consider a binary tree as first-level structure. With each node v of the tree, we associate a line $bl(v)$ (standing for *base line* of v) and the set of segments intersected by the line.

More formally, let N be a set of NCT segments. We order the set of endpoints corresponding to such segments in ascending order according to their x -values. Then, we determine a vertical

line partitioning such ordered set in two subsets of equal cardinality and we associate such vertical line with the base line $bl(r)$ of the root. All segments intersecting $bl(r)$ are associated with the root whereas all segments which are on the left (right) of $bl(r)$ and do not intersect it, are passed to the left (right) subtree of the root. Note that the number of segments passed to any of subtrees is at most half the number of segments stored in the root. The decomposition recursively continues until each leaf node contains B segments and fits as a whole in internal memory. The construction of base lines guarantees that the segments in a node v are intersected by $bl(v)$ but are not intersected by the base line of the parent of v . The tree height is $O(\log n)$.

Second-level data structures. Because of the above construction, each segment in an internal node v either lies on $bl(v)$ or intersects it. The segments which lie on the base line are stored in $C(v)$, an external interval tree [3] which requires a linear number of storage blocks and performs a VS query in $O(\log_B n + t)$ I/O's.

Each segment which is intersected by $bl(v)$ has left and right parts. Left and right parts of all the segments are collected into two sets, called $L(v)$ and $R(v)$, respectively. Each of these sets contains line-based segments and can be efficiently maintained in secondary storage using the technique proposed in Section 2. Totally, each segment is represented at most twice inside the two-level data structure. Therefore, the tree stores N segments in $O(n)$ blocks in secondary storage.

Figure 4 (b) illustrates the organization and content of the proposed 2LDS, for the set of segments presented in Figure 4 (a).

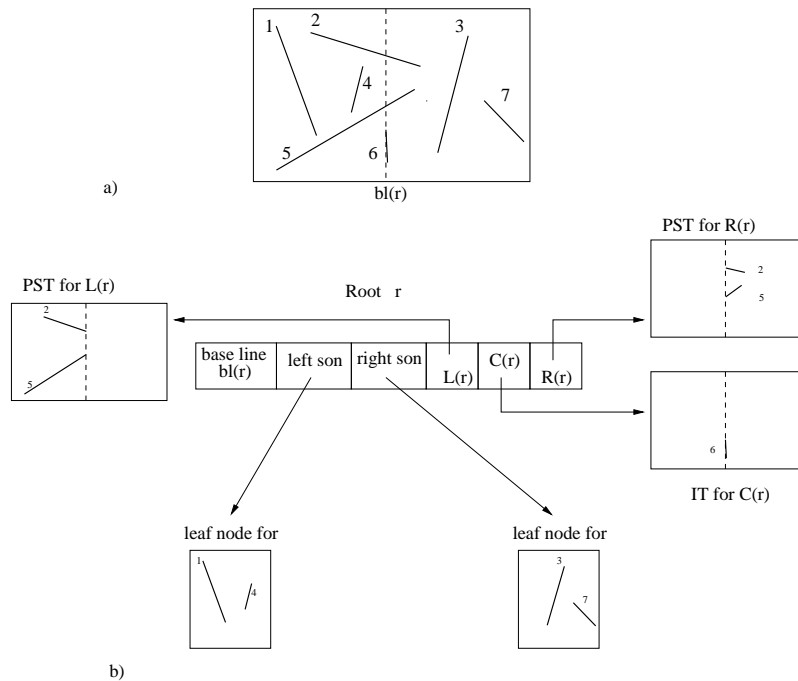


Figure 4: (a) A set of 7 NCT segments; (b) The corresponding data structure ($B = 2$, PST stands for priority search tree, IT stands for interval tree).

Search algorithm. Given a query segment of the form $x = x_0, a \leq y \leq b$, the search is performed on the first-level tree as follows. We scan the tree and visit exactly one node v for each level. In each node v , we first verify if x_0 equals the x -coordinate of the vertical line $bl(v)$. In such a case, all segments in $C(v)$, $L(v)$ and $R(v)$ intersected by q are retrieved and the search stops. Otherwise, if x_0 is lower than the x -coordinate of $bl(v)$, we visit only $L(v)$

and move to the left son of v . If x_0 is greater than the x -coordinate of $bl(v)$, we visit $R(v)$ and move to the right son. The search for all segments T' inside one node intersected by the query requires $O(\log_B n + IL^*(B) + \frac{T'}{B})$ time. Since the height of the first-level data structure is $O(\log n)$ and each segment is reported only once, the I/O complexity of the total search is $O(\log n(\log_B n + IL^*(B)) + t)$.

Updates. If updates are allowed, the binary tree should be replaced by a dynamic search-tree, for which efficient rebalancing methods are known. To maintain insertions and deletions of line-based segments in the data structure described above, we replace the binary tree with a $BB[\alpha]$ -tree [5, 15], $0 < \alpha < 1 - 1/\sqrt{2}$. We store balance values in internal nodes of the $BB[\alpha]$ -tree and maintain the optimal $O(\log n)$ height of the tree performing $O(\log n)$ single or double rotations during an update. The update cost consists of $O(\log n)$ operations for the search and balance maintenance in the first-level tree and $O(\log_B n + \frac{\log_B^2 n}{B})$ operations for updating the second-level data structures. Therefore, the total update cost is $O(\log n + \frac{\log_B^2 n}{B})$. The cost is $O(\log n)$ for all real values of n (more exactly, for $n \in O(2^B)$).

The results stated by the next theorem follow from the previous discussion.

Theorem 1 N NCT segments can be stored in a secondary storage data structure having the following costs: (i) storage cost is $O(n)$; (ii) VS query time is $O(\log n(\log_B n + IL^*(B)) + t)$; (iii) update time is $O(\log n + \frac{\log_B^2 n}{B})$. \square

4 An improved solution to query NCT segments

In order to improve the complexity results obtained in the previous section, a secondary storage interval tree, designed for solving stabbing queries [3], is used as first-level data structure, instead of the binary tree. This modification, together with the use of the fractional cascading technique [4], improves the wasteful factor $\log n$ in the complexity results presented in Theorem 1, but using $O(n \log_2 B)$ space.

4.1 First-level data structure

The *interval tree* is a standard dynamic data structure for storing a set of 1-dimensional segments [8], tailored to support stabbing queries. The external-memory interval tree is based on a *buffer tree* technique [1]. The tree is balanced over the segment endpoints, has a branching factor b , and require $O(n)$ blocks for storage. Segments are stored in secondary structures associated with the internal nodes of the tree. As first level data structure, we use an external-memory interval tree and we select b equal to $B/4$. The height of the first-level structure is $O(\log_b n) = O(\log_B n)$. The first level of the tree partitions the data into $b + 1$ slabs separated by vertical lines s_1, \dots, s_b . In Figure 5, such lines are represented as dashed lines. In the example, b is equal to 5. Multislabs are defined as contiguous ranges of slabs such as, for example, $[1 : 4]$. There are $O(b^2)$ multislabs in each internal node. Segments stored in the root are those which intersect one or more dashed lines s_i . Segments that intersect no line are passed to the next level (segments 3, 4 and 7 in Figure 5). All segments between lines s_{i-1} and s_i are passed to the node corresponding to the i -th slab. The decomposition continues until each leaf represents B segments.

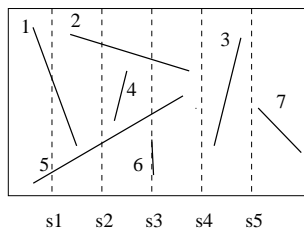


Figure 5: Partition of the segments by lines s_i .

4.2 Second-level data structures

In each internal node of the first-level tree, we split all segments which do not lie on dashed lines s_i into *long* and *short* fragments. A long fragment spans one or more slabs and has both its endpoints on dashed lines. A short fragment spans no complete slab and has only one endpoint on the dashed line. Segments are split as follows (see Figure 6). If a segment completely spans one or more slabs, we split it into one long (central) fragment and at most two short fragments. The long fragment is obtained by splitting the segment on the boundaries of the largest multislab it spans. After this splitting, at most two additional short segments are generated. If a segment in the node intersects only one dashed line and spans no slab, it is simply split into two short fragments. In total, if k segments are associated with a node, the splitting process generates at most k long and $2k$ short fragments.

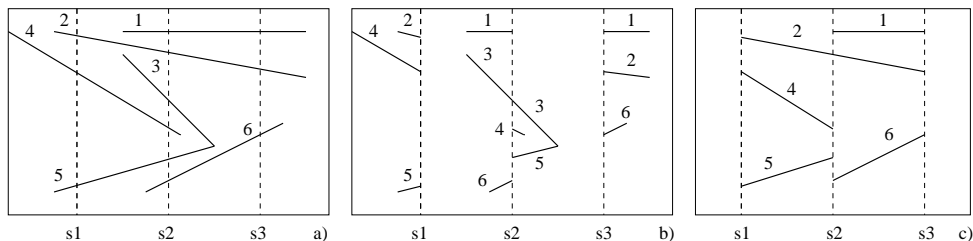


Figure 6: The splitting of segments. a) Segments associated with a node; b) short fragments; c) long fragments.

As before, segments lying on a dashed line s_i are stored in an external interval tree C_i . Short and long fragments are stored as follows.

Short fragments. All short fragments are naturally clustered according to the dashed line they touch. Note that short fragments having one endpoint on line s_i are line-based segments and can be maintained in an external priority search tree as described in Section 2. Short line-based fragments which are located on the left of s_i are stored in an external PST L_i . Symmetrically, short fragments on the right side of s_i are stored in an external PST R_i . Totally, an internal node of the first-level structure contains $2b$ external PSTs for short fragments.

Long fragments. We store all long segments in an additional structure G which is essentially a segment tree [1, 6] based on dashed lines s_i , $i = 1, \dots, b$. G is a balanced binary tree with $b - 2$ internal nodes and $b - 1$ leaves. Thus, in total it has $O(B)$ nodes.

Each leaf of the segment tree G corresponds to a single slab and each internal node v is associated with the multislab $I(v)$ formed by the union of the slabs associated with the leaves of the subtree of v . The root of G is associated with the multislab $[1 : b]$. Given a long fragment l which spans many slabs, the *allocation nodes* of l are the nodes v_i of G such that l spans $I(v_i)$ but not $I(\text{par}(v_i))$, where $\text{par}(v)$ is the parent of v in G . There are at most two allocation nodes

of l at any level of G , so that, since the height of the segment tree is $\log_2 B$, l has $O(\log_2 B)$ allocation nodes [6].

Each internal node v of G is associated with a multislab $[i : j]$ and is associated with the ordered list (called *multislab list* $[i : j]$) of long fragments having v as allocation node, cut on the boundaries of $I(v)$. A B^+ -tree is maintained on the list for fast retrieval and update.

Since the segment tree G contains $O(B)$ nodes, each containing a pointer to a B^+ -tree in addition to standard node information, it can be stored in $O(1)$ blocks. In total, each segment may be stored in at most three external-memory structures. That is, if a segment spans the multislab $[i : j]$, the segment is stored in data structures L_i, R_j and it has $O(\log_2 B)$ allocation nodes of G . Since $b = B/4$, an internal node of the first-level structure has enough room to store all references to b structures C_i , b structures L_i , b structures R_i and one structure G . Thus, in total, the space utilization is $O(n \log_2 B)$.

Search algorithm. Given a query segment $x = x_0, a_1 \leq y \leq a_2$, a lookup is performed on the first-level tree from the root, searching for a leaf containing x_0 . For each node, if x_0 equals the x -coordinate of any s_i , the interval tree C_i is searched together with the second-level structures R_i and L_i to retrieve the segments lying on s_i and short fragments intersected by the query segment. Otherwise, if x_0 hits the slab i , that is $s_i < x_0 < s_{i+1}$, then we check second-level structures R_i and L_{i+1} .

In both cases, we have to check also the second-level structures G which contain multislabs spanning the query value x_0 and retrieve all the long fragments intersected by the query. When visiting G , we scan from the root of G to a leaf containing the value x_0 . In each visited node, we search the ordered list associated with the node. Finally, if x_0 does not coincide with any s_i in the node, the search continues on the next level, in the node associated with the slab containing x_0 .

Although any segment may be stored in three different external structures, it is clear that each segment intersected by the query q is retrieved only once. Moreover, for each internal node, during the search we visit exactly two structures for short fragments and structure G for long ones. This proves the following lemma.

Lemma 4 N NCT segments can be stored in a secondary storage data structure having the following costs: (i) storage cost is $O(n \log_2 B)$; (ii) VS query time is $O(\log_B n (\log_B n \log_2 B + IL^*(B)) + t)$. \square

To further reduce the search time, we extend this approach with the technique of fractional cascading [4] between lists stored on neighbor levels of structures G .

4.3 Fractional cascading

Fractional cascading [4] is a technique supporting the execution of a sequence of searches at a constant cost (in both internal memory and secondary storage) per search, except for the first one. The main idea is to construct a number of “bridges” among lists. Once an element is found in one list, the location of the element in other lists is quickly determined by traversing the bridges rather than applying the general search. Fractional cascading has been extensively used in internal-memory algorithms [4, 6, 10]. Recently, the technique has been also applied to off-line external-memory algorithms [2]. Our approach can be summarized as follows.

Data structure supporting fractional cascading. The idea is to create bridges between nodes on neighbor levels of the G structure, stored in one node of the first-level data structure.

In particular, for an internal node of G associated with a multislab $[i : j]$, two sets of bridges are created between the node and its two sons associated with multislabs $[i : \frac{i+j}{2}]$ and $[\frac{i+j}{2} : j]$ (see Figure 7). Each fragment in the multislab list $[i : j]$ keeps two references to the nearest fragments in the list, which are bridges to left and right sons.

For multislabs $[i : j]$ and $[i : \frac{i+j}{2}]$, the bridges are created in such a way that the following d -property is satisfied : *the number s of fragments in both multislabs $[i : j]$ and $[i : \frac{i+j}{2}]$ between two sequential bridges is always such that $d \leq s \leq 2d$, where d is a constant ≥ 2 .*

The bridges between two multislab lists $[i : j]$ and $[i : \frac{i+j}{2}]$ are generated as follows. First we merge the two lists in one. All fragments in the joined list do not intersect each other and either touch or intersect the line $s_{\frac{i+j}{2}}$. We scan the joined list given by the order of segment intersections with line $s_{\frac{i+j}{2}}$ and select each $d + 1$ -th fragment from the list as a bridge. If the fragment is from $[i : j]$ (like fragment 7 on Figure 7), we cut it on line $s_{\frac{i+j}{2}}$ and copy it in the multislab list $[i : \frac{i+j}{2}]$. Such a copy of the bridge is called *augmented bridge fragment*; in Figure 7 these fragments are marked with “*”⁶. Otherwise, if the fragment is from $[i : \frac{i+j}{2}]$ (like fragment 4 on Figure 7), we copy it in the multislab list $[i : j]$. The position of the augmented bridge fragment in $[i : j]$ is determined by its intersection with line $s_{\frac{i+j}{2}}$. Analogously, the bridges are created between multislabs $[i : j]$ and $[\frac{i+j}{2} : j]$. Bridge fragments from a multislab list $[i : j]$ are copied (after the cutting) in the multislab list $[\frac{i+j}{2} : j]$ while bridge fragments from the multislab list $[\frac{i+j}{2} : j]$ are copied to $[i : j]$.

After bridges from the multislab list $[i : j]$ to both lists $[i : \frac{i+j}{2}]$ and $[\frac{i+j}{2} : j]$ are generated, the list $[i : j]$ contains original fragments (some of them are bridges to left or right son) and augmented bridge fragments copied from lists $[i : \frac{i+j}{2}]$ and $[\frac{i+j}{2} : j]$. In Figure 7, the list $[i : j]$ contains three augmented bridge fragments, respectively fragments 3, 4, and 9. All the fragments in $[i : j]$ are ordered by the points they intersect or touch the line $s_{\frac{i+j}{2}}$.

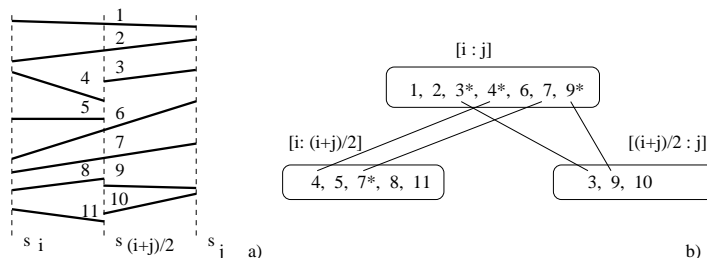


Figure 7: “Bridges” in G . a) Long fragments stored in the node associated with multislab $[i : j]$ and in its two sons, associated with multislabs $[i : \frac{i+j}{2}]$ and $[\frac{i+j}{2} : j]$. b) Lists of fragments associated with nodes of the G structure. The lists are extended with bridge segments ($d=2$). Bridges are shown by lines and bridge elements are marked with ‘*’.

Given an internal node v of G , associated with multislab $[i : j]$, a B^+ -tree is built from the multislab list $[i : j]$, after bridges to both sons are generated and copied in the list. In the full paper we prove that, after including augmenting lists in all nodes of G , the space complexity is still $O(n \log_2 B)$.

Search algorithm. Let q be the vertical segment of the form $x = x_0, a_1 \leq y \leq a_2$. The VS query q is performed as follows. We search the first-level data structure from the root down to

⁶Note that augmented bridge fragments are only used to speed up the search, they are never reported in the query reply.

a leaf containing x_0 . In each node of the first-level structure, we first detect the slab i which is hit by x_0 . As before, we search two second-level structures L_i and R_{i+1} for short fragments intersected by the query.

To retrieve long fragments intersected by the query in the node of the first-level structure, we search in G . First we search in the B^+ -tree associated with the root of G and detect the leftmost segment fragment f_l^1 intersected by q and associated with the root. This takes $O(\log_B n)$ steps. Then, the leaves of the B^+ -tree are traversed and all fragments in the root of G intersected by q (except the augmented bridge fragments) are retrieved.

As a second step, if x_0 is lower than $s_{\frac{i+j}{2}}$, the bridge to the left son, nearest to f_l^1 , is determined, otherwise the bridge to the right son, nearest to f_l^1 , is determined. Following the appropriate bridge, a leaf node in the B^+ -tree associated with a second level node of G is reached. Because of the d -property of bridges, the leftmost segment fragment f_l^2 , contained in the reached leaf node and intersected by q , can be found in $O(1)$ I/O's. Then, the leaves of the B^+ -tree are traversed and all fragments intersected by q (except the augmented bridge fragments) are retrieved. The same procedure of bridge navigation and fragment retrieval is repeated on levels $3, \dots, \log_2 b$ of G .

With the use of bridges, searching for the leftmost fragment intersecting q on all levels of G takes $O(\log_B n + \log_2 B)$ steps. Together with searching in L_i and R_{i+1} for short fragments, the search time for one internal node a of the first-level structure is $O(\log_B n + \log_2 B + IL^*(B) + \frac{T'}{B})$, where T' is the number of segments in node a intersected by the query. Since any segment is stored in only one node of the first-level tree (whose height is $O(\log_B n)$) and each segment intersected by the query is reported only once, reporting all segments intersected by the query takes $O(\log_B n (\log_B n + \log_2 B + IL^*(B)) + t)$.

Insertions. The 2LDS proposed above has been designed for the static case. The extension of the proposed schema to the semi-dynamic case is based on: (i) the use of a weighted-balanced B-tree [3] as first-level data structure;⁷ (ii) the use of a $BB[\alpha]$ -tree [5, 15], $0 < \alpha < 1 - 1/\sqrt{2}$ as the second-level structure G for long fragments; (iii) the definition of some additional operations on multislab lists (similar to those presented in [10]), guaranteeing the $O(1)$ I/O amortized complexity of bridge navigation. Such extensions allow to execute insertions in $O(\log_B n + \log_2 B + \frac{\log_B^2 n}{B})$ amortized time.

Theorem 2 N NCT segments can be stored in a secondary storage data structure having the following costs: (i) storage cost is $O(n \log_2 B)$; (ii) VS query time is $O(\log_B n (\log_B n + \log_2 B + IL^*(B)) + t)$; (iii) insertion amortized time is $O(\log_B n + \log_2 B + \frac{\log_B^2 n}{B})$. \square

5 Concluding remarks

In this paper we have proposed two techniques to solve a vertical (or having any other fixed direction) segment query on segment databases. The more efficient technique has $O(n \log_2 B)$ space complexity and time complexity very close to $O(\log_B^2 n + t)$. Future work includes the extension of the proposed technique to deal with query segments having arbitrary angular coefficients.

⁷A weighted-balanced B-tree is a B-tree where the weight of a node (i.e., the number of elements stored in the subtree rooted at the node) is a function of the level to which the node belongs. It guarantees efficient rebalancing operations in external-memory.

References

- [1] L. Arge. The Buffer Tree: A New Technique For Optimal I/O Algorithms. In *LNCS 955: Proc. of the 4th Int. Workshop on Algorithms and Data Structures*, pages 334-345, 1995.
- [2] L. Arge, D.E. Vengroff, and J. S. Vitter. External-Memory Algorithms for Processing Line Segments in Geographic Information Systems. In *Proc. of the 3rd Annual European Symp. on Algorithms*, pages 295-310, 1995.
- [3] L. Arge and J. S. Vitter. Optimal Dynamic Interval Management in External Memory. In *Proc. of the Int. Symp. on Foundations of Computer Science*, pages 560-569, 1996.
- [4] B. Chazelle and L.J. Guibas. Fractional cascading : I. A Data Structuring Technique. *Algorithmica*, 1(2):133-162, 1986.
- [5] S.W. Cheng and R. Janardan. Efficient dynamic algorithms for some geometric intersection problems. *Information Processing Letters*, 36(5):251-258, 1990.
- [6] Y.-J. Chiang and R. Tamassia. Dynamic Algorithms in Computational Geometry. *Proc. IEEE*, 80(9):1412-1434, 1992.
- [7] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2), pages 121-138, 1979.
- [8] H. Edelsbrunner. A New Approach to Rectangular Intersection. Part I. *Intern. J. Comp. Mathematics*, 13:209-219, 1983.
- [9] C. Icking, R. Klein, and T. Ottmann. Priority Search Trees in Secondary Memory. In *LNCS 314: Proc. of the Int. Workshop on Graph-Theoretic Concepts in Computer Science*, pages 84-93, 1988.
- [10] K. Mehlhorn and S. Naher. Dynamic Fractional Cascading. *Algorithmica*, 5, pages 215-241, 1990.
- [11] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *J. Comp. System Sciences*, 51(1):25-52, 1995.
- [12] P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, and J.S. Vitter. Indexing for Data Models with Constraints and Classes. *J. Comp. System Sciences*, 52(3):589-612, 1996.
- [13] M. Koubarakis. Database Models for Infinite and Indefinite Temporal Information. *Information Systems*, 19(2):141-173, 1994.
- [14] E. McCreight. Priority Search Trees. *SIAM Journal of Computing*. 14(2): 257-276, 1985.
- [15] J. Nievergelt and E. M. Reingold. Binary Search Tree of Bounded Balance. *SIAM J. Computing*, 2(1):33-43, 1973.
- [16] J. Paredaens. Spatial Databases, the Final Frontier. In *LNCS 893: Proc. of the 5th Int. Conf. on Database Theory*, pages 14-31, 1995.
- [17] S. Ramaswamy and S. Subramanian. Path-Caching: A Technique for Optimal External Searching. In *Proc. of the ACM Symp. on Principles of Database Systems*, pages 25-35, 1994.

- [18] S. Ramaswamy. Efficient Indexing for Constraints and Temporal Databases. In *LNCS 1186: Proc. of the 6th Int. Conf. on Database Theory*, pages 419-431, 1997.
- [19] S. Subramanian and S. Ramaswamy. The P-Range Tree: A New Data Structure for Range Searching in Secondary Memory. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms*, pages 378-387, 1995.

A Some relevant algorithms

```

Algorithm 1
input : a PST  $T$  for a set of  $N$  line-based segments
        a query segment  $q$ 
output : the deepest-leftmost segment intersected by the query segment  $q$ , with respect to its storage position in  $T$ 
         the node in  $T$  where the segment is located
begin
let  $q$  be the segment  $q.l \leq x \leq q.r, y = q.y$ 
let  $(v.left.x, v.left.y)$  be the upper endpoint of segment  $v.left$ 
let  $(v.right.x, v.right.y)$  be the upper endpoint of segment  $v.right$ 
let  $Q$  be the empty queue
Initialize  $Q$  with the tree root
 $answer_n \leftarrow \infty; answer_s \leftarrow \infty;$ 
repeat
  Extract a block  $v$  from the front of  $Q$ 
  if some segments in  $v$  intersect  $q$  then
    select the leftmost segment in  $v$  and  $answer_s$  and assign it to  $answer_s$ 
    select the block of  $Tr$  containing  $answer_s$  and assign it to  $answer_n$ 
     $Q$  is updated according to the following cases:
    if  $q.y \geq v.low$  then
      no segments in the blocks having  $v$  as ancestor in  $Tr$  can intersect  $q$  and the queue remains unchanged
    else
      a) if  $q.y > v.left.y$  and  $q.y > v.right.y$  then  $Q$  remains unchanged (Figure 10.a) endif
      b) if  $q.y \leq v.left.y$  and  $q.l$  is on the left of  $v.left$  then the left son of  $v$  is added at the end of  $Q$  (Figure 10.b). In
         symmetric case if  $q.y \leq v.right.y$  and  $q.l$  is on the right of  $v.right$ , the right son of  $v$  is added at the end of  $Q$ 
         endif
      c) if  $q.y \leq v.left.y, q.y > v.right.y$  and  $q.l$  lies between  $v.left$  and  $v.right$  then the left son of  $v$  is added at the end
         of  $Q$  (Figure 10.c). Symmetric case is treated similarly endif
      d) if  $q.y \leq v.left.y, q.y \leq v.right.y$  and  $q.l$  lies between  $v.left$  and  $v.right$  then we empty queue  $Q$  and then insert both
         sons in it (Figure 10.d) endif
    endif
  until  $Q$  is empty
return  $answer_s$  and  $answer_n$ 
end

```

Figure 8: Function Find.

```

Algorithm 2
input : a PST  $T$  for a set of  $N$  line-based segments
        a segment query  $q$ 
output : all segments stored in  $Tr$  and intersected by  $q$ 
begin
let  $q$  be the segment  $q.l \leq x \leq q.r, y = q.y$ 
Apply function Find to  $Tr$  and  $q$ 
let  $s_l$  and  $bl_l$  be the segment and the block containing  $s_l$  located by function Find, respectively
if  $bl_l = \infty$  then  $q$  does not intersect any segment
else
  Apply the symmetric version of function Find ( $Find'$ ) to  $Tr$  and  $q$ ,
  retrieving the rightmost segment intersected by  $q$  and the node where it is contained
  let  $s_r$  and  $bl_r$  be the segment and the block containing  $s_r$  located by function  $Find'$ , respectively
  let  $lca$  be the lowest common ancestor of  $bl_l$  and  $bl_r$  in  $Tr$ 
  let  $P_l$  and  $P_r$  be the paths from  $bl_l$  to  $lca$  and from  $bl_r$  to  $lca$ , respectively
  Walk up  $P_l$ 
  for each node  $v$  in  $P_l$  do
    retrieve all segments in  $v$  intersected by  $q$ 
    if (case 1)  $v = bl_l$  or (case 2)  $v \neq lca$  and the predecessor of  $v$  on  $P_l$  is a left child of  $v$  then
      if (case 1) then  $z = v$  else (case 2)  $z = v$ 's right child endif
      perform a preorder traversal of the sub-tree rooted at  $z$ 
      for each visited node  $w$  do
        retrieve all segments in  $w$  which intersect  $q$ 
        if  $w.low < q.y$  or  $w$  does not contain any segment intersecting  $q$  then
          do not proceed the traversal in  $w$ 's children endif
      endfor
    endif
  endfor
  The above steps are repeated also on  $P_r$ , with "left" and "right" interchanged
  let  $\bar{P}$  be the path from  $lca$  to the root of the tree
  Walk up  $\bar{P}$ 
  for each node  $v$  in  $\bar{P}$  do
    retrieve all segments in  $v$  intersected by  $q$ 
  endfor
endif
end

```

Figure 9: Function Report.

B Some additional figures

Figure 10 illustrates some different cases, considered in function *Find* (see Appendix A). Figure 11 shows all nodes visited by the *Report* algorithm.

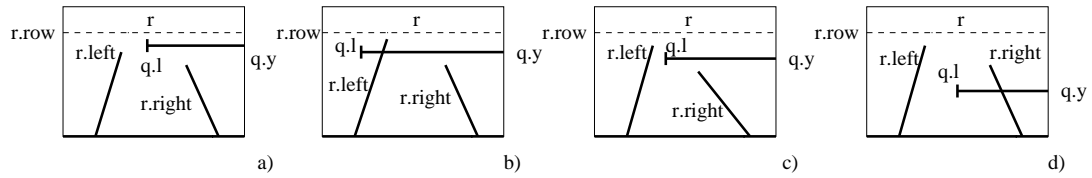


Figure 10: Different cases in function *Find*.

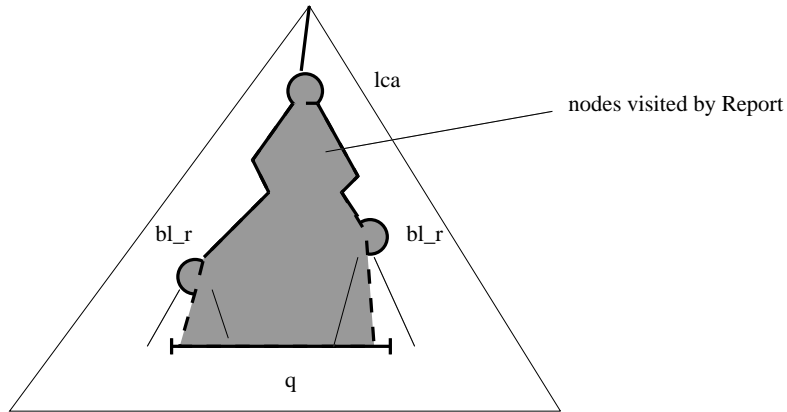


Figure 11: Search space visited by the *Report* algorithm.