I/O-efficient Point Location using Persistent B-Trees

Lars Arge* Andrew Danner† Sha-Mayn Teh

Department of Computer Science
Duke University

Abstract

We present an external planar point location data structure that is I/O-efficient both in theory and practice.

The developed structure uses linear space and answers a query in optimal $O(\log_B N)$ I/Os, where B is the disk block size. It is based on a persistent B-tree, and all previously developed such structures assume a total order on the elements in the structure. As a theoretical result of independent interest, we show how to remove this assumption.

Most previous theoretical I/O-efficient planer point location structures are relatively complicated and have not been implemented. Based on a bucket approach, Vahrenhold and Hinrichs therefore developed a simple and practical, but theoretically non-optimal, heuristic structure. We present an extensive experimental evaluation that shows that on a range of real-world Geographic Information Systems (GIS) data, our structure uses fewer I/Os than the structure of Vahrenhold and Hinrichs to answer a query. On a synthetically generated worst-case dataset, our structure uses significantly fewer I/Os.

1 Introduction

The planar point location problem is the problem of storing a planar subdivision defined by N segments such that the region containing a query point p can be computed efficiently. Planar point location has many applications in, e.g., Geographic Information Systems (GIS), spatial databases, and graphics. In many of these applications, the datasets are larger than the size of physical memory and must reside on disk. In such cases, the transfer of data between the fast main memory and slow disks (I/O), not the CPU computation time, often becomes the bottleneck. Therefore, we are interested in planar point location structures that minimize the number of I/Os needed to answer a query.

While several theoretically I/O-efficient planar point location structures have been developed, e.g., [17, 1, 8], they are all relatively complicated and consequently none of them have been implemented. Based on a bucket approach, Vahrenhold and Hinrichs developed a simple and practically efficient, but theoretically non-optimal, heuristic structure [24]. In this paper, we show that a point location structure based on a persistent B-tree is efficient both in theory and practice; the structure obtains the theoretical optimal bounds and our experimental investigation shows that, for a wide range of real-world GIS data, it uses fewer I/Os to answer a query than the structure of Vahrenhold and Hinrichs. For a synthetically generated worst case dataset, the structure uses significantly less I/Os to answer a query.

1.1 I/O-model and previous results

We work in the standard I/O model of computation proposed by Aggarwal and Vitter [2]. In this model, computation can only occur on data stored in a main memory of size M, and an I/O transfers a block of B consecutive elements between an infinite sized disk and main memory. The complexity measures in this model are the number of I/Os used to solve a problem (answer a query) and the number of disk blocks used.

Aggarwal and Vitter showed that $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ is the external memory equivalent of the well-known $O(N\log_2 N)$ internal memory sorting bound. Similarly, the B-tree [10, 12, 18] is the external equivalent of an internal memory balanced search tree. It uses linear space, $O(N/B)$ blocks, to store N elements; supports updates in $O(\log_B N)$ I/Os; and performs one dimensional range queries in optimal $O(\log_B N + T/B)$ I/Os, where T is the number of reported elements. Using a general technique by Driscoll et al. [14], persistent versions of the B-tree have also been developed [11, 26]. A persistent data structure maintains a history of all updates performed on it, such that queries can be answered on any of the previous versions of the structure, while updates can only be performed on the most recent version (thus creating a new version).[1] A persistent B-tree uses $O(N/B)$ space, where N is the number of updates performed, and updates and range queries can be performed in $O(\log_B N)$ and $O(\log_B N+T/B)$ I/Os,

*Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA–9972879, CAREER grant CCR–9984099, ITR grant EIA–0112849, and U.S.-Germany Cooperative Research Program grant INT–0129182. Email: large@cs.duke.edu.

†Supported in part by the National Science Foundation through grant CCR–9984099. Email: adanner@cs.duke.edu.

[1] The type of persistence we describe here is often called *partially persistent* as opposed to *full persistence* where updates can be performed on any previous version.

respectively [11, 26]; note that the structure requires that all elements stored in it during its entire lifespan are comparable, that is, that the elements are totally ordered.

In the RAM model, several linear space planar point location structures that can answer a query in optimal $O(\log_2 N)$ time have been developed (e.g [16, 20, 19]). One of these structures, due to Sarnak and Tarjan [20], is based on a persistent search tree. In this application of persistence, not all elements (segments) stored in the structure over its lifespan are comparable; thus a similar I/O-efficient structure cannot directly be obtained using a persistent B-tree. See the recent survey by Snoeyink [21] for a full list of RAM model results.

In the I/O-model, Goodrich et al. [17] developed an optimal static point location structure using linear space and answering a query in $O(\log_B N)$ I/Os. Agarwal et al. [1] and Arge and Vahrenhold [8] developed dynamic structures. Several structures for answering a batch of queries have also been developed [17, 9, 13, 24]. Refer to [4] for a survey. While these structures are all theoretically I/O-efficient, they are all relatively complicated and consequently none of them have been implemented. Based on an internal memory bucket approach [15], Vahrenhold and Hinrichs therefore developed a simple but non-optimal heuristic structure, which performs well in practice [24]. The main idea in this structure is to impose a grid on the segments defining the subdivision and store each segment in a "bucket" corresponding to each grid cell it intersects. The grid is constructed such that for certain kinds of "nice data", each segment is stored in $O(1)$ buckets (such that the structure requires linear space) and each bucket contains $O(B)$ segments (such that a query can be answered in $O(1)$ I/Os). In the worst case however, each segment may be stored in $\Theta(\sqrt{N/B})$ buckets and consequently the structure may use $\Theta(N/B\sqrt{N/B})$ space. In this and some other cases, there may be a bucket containing $O(N)$ segments such that a query takes $O(N/B)$ I/Os.

Most of the structures in the above results actually solve a slightly generalized version of the planar point location problem, namely the *vertical ray-shooting problem*: Given a set of N non-intersecting segments in the plane, the problem is to construct a data structure such that the segment directly above a query point p can be found efficiently. This is also the problem we consider in this paper.

1.2 Our results

The main result of this paper is an external data structure for vertical ray-shooting (and thus planar point location) that is I/O-efficient both in theory and practice. The structure, described in Section 2, uses linear space and answers a query in optimal $O(\log_B N)$ I/Os. It is based on the persistent search tree idea of Sarnak and Tarjan [20], and as a theoretical contribution of independent interest, we show how to modify the known persistent B-tree such that only elements present in the same version of the structure need to be comparable, that is, so no total order is needed. In Section 3, we then present an extensive experimental evaluation of the structure's practical performance compared to the heuristic grid structure of Vahrenhold and Hinrichs using both real-world and artificial (worst case) datasets. In their original experimental evaluation, Vahrenhold and Hinrichs [24] used hydrology and road feature data extracted from the U.S. Geological Survey Digital Line Graph dataset [23]. On similar "nicely" distributed sets of short segments, our structure answers queries in about half as many I/Os as the grid structure but requires about twice as much space. On less "nice" data, our structure performs significantly better than the grid structure; we present one example where our structure answers queries using 90% fewer I/Os and requires 94% less space.

2 Ray-shooting using persistent B-trees

Our structure for answering vertical ray-shooting queries among a set of non-intersecting segments in the plane is based on the persistent search tree idea of Sarnak and Tarjan [20]. This idea utilizes the fact that any vertical line l in the plane naturally introduces an "above-below" order on the segments it intersects. This means that if we conceptually sweep the plane from left to right ($-\infty$ to ∞) with a vertical line, inserting a segment in a persistent search tree when its left endpoint is encountered and deleting it again when its right endpoint is encountered, we can answer a ray-shooting query $p = (x, y)$ by searching for the position of y in the version of the search tree we had when l was at x. Refer to Figure 1. Note that two segments that cannot be intersected with the same vertical line are not "above-below" comparable. This

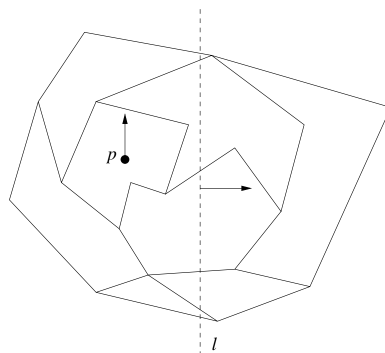


Figure 1: Vertical ray-shooting using sweep and persistent search tree.

means that not all elements (segments) stored in the persistent structure over its lifespan are comparable and thus an I/O-efficient structure cannot directly be obtained using a persistent B-tree. To make the structure I/O-efficient, we need a persistent B-tree that only requires elements present in the same version of the structure to be comparable. In Section 2.1, we first sketch the persistent B-tree of [11] and then in Section 2.2 we describe the modifications needed to use the tree in a vertical ray-shooting structure.

2.1 Persistent B-tree

A B-tree, or more generally an (a,b)-tree [18], is a balanced search tree with all leaves on the same level, and with all internal nodes except possibly the root having $\Theta(B)$ children (typically between $B/2$ and B). Normally, elements are stored in the leaves, and the internal nodes contain "routing elements" used to guide searches (sometimes called a B^+-tree). Since each node and leaf contains $\Theta(B)$ elements it can be stored in $O(1)$ blocks, which in turn means that the tree uses linear space ($O(N/B)$ disk blocks). Since the tree has height $O(\log_B N)$, a range search can be performed in $O(\log_B N + T/B)$ I/Os. Insertions and deletions can be performed in $O(\log_B N)$ I/Os using *split*, *merge*, and *share* operations on the nodes on a root-leaf path [18, 10, 12].

A persistent (or *multiversion*) B-tree as described in [11, 26] is a directed acyclic graph (DAG) with elements in the sinks (leaves) and "routing elements" in internal nodes. Each element is augmented with an insert and a delete *version* (or *time*), defining the *existence interval* of the element; an element is *alive* in its existence interval and *dead* otherwise. Similarly, an existence interval is associated with each node, and it is required that the nodes and elements alive at any time t (in version t) form a B-tree with fanout between αB and B for some constant $0 < \alpha < 1/2$. Given the appropriate root (in-degree 0 node) we can thus perform a range search in any version of the structure in $O(\log_B N + T/B)$ I/Os. To be able to find the appropriate root at time t in $O(\log_B N)$ I/Os, the roots are stored in a standard B-tree, called the *root B-tree*.

An update in the current version of a persistent B-tree (and thus the creation of a new version) may require structural changes and creation of new nodes. To control these changes and obtain linear space use, an additional invariant is imposed on the structure: Whenever a new node is created, it must contain between $(\alpha + \gamma)B$ and $(1 - \gamma)B$ alive elements (and no dead elements) for $0 < \gamma < 1/2 - \alpha/2$.

To insert a new element x in the current version of a persistent B-tree we first perform a search for the relevant leaf l using $O(\log_B N)$ I/Os. Then we insert x in l. If l now contains more than B elements, we have a *block overflow*. In this case we perform a *version-split*; we copy all, say k, alive elements from l and mark l as deleted, i.e, we update its existence interval to end at the current time. If $(\alpha + \gamma)B \le k \le (1 - \gamma)B$, we create a new leaf with the k elements and recursively update *parent*(l) by persistently deleting the reference to l and inserting a reference to the new node. If on the other hand $k < (\alpha + \gamma)B$ or $k > (1 - \gamma)B$, we have a *strong underflow* or *strong overflow*, respectively. The strong overflow case is handled using a *split*; we simply create two new nodes with approximately half of the k elements each, and update *parent*(l) recursively in the appropriate way. To guarantee that a split cannot result in a strong underflow we require that $\frac{1}{2}(1 - \gamma)B + 1 > (\alpha + \gamma)B$ or equivalently that $B < 2\alpha + 3\gamma - 1$. Note the similarity with a split rebalancing operation on a normal B-tree. Similarly, the strong underflow case is handled with operations similar to merge and share rebalancing operations on normal B-trees; we perform a version split on a sibling l' of l to obtain $k' > \alpha B$ other alive elements. To guarantee that we now have enough alive elements for a new node we require that $k + k' > 2 \cdot \alpha B - 1 > (\alpha + \gamma)B$ or equivalently that $B > 1/(\alpha - \gamma)$. If $k + k' \le (1 - \gamma)B$, we create a new leaf with the $k + k'$ elements. If on the other hand $k + k' > (1 - \gamma)B$, we first perform a split in order to create two new leaves. The first case corresponds to a *merge* and the second to a *share*. Finally, we recursively update *parent*(l) appropriately.

A deletion is handled similarly to an insertion; first the relevant element x in a leaf l is found and marked as deleted. This may result in l containing less than αB alive elements, also called a *block underflow*. To reestablish the invariants we simply perform a version-split followed by a merge (and possibly a split as before). Figure 2 illustrates the "rebalance operations" needed as a result of an insertion or deletion.

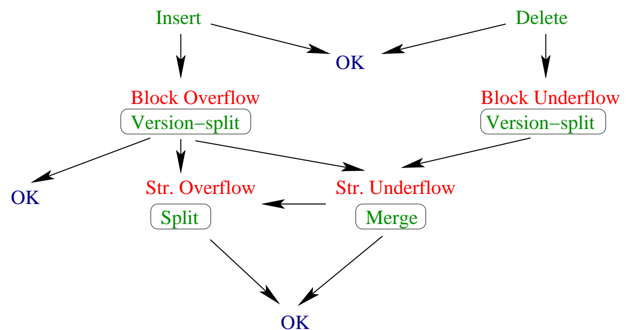


Figure 2: Illustration of "rebalancing operations" needed when updating a persistent B-tree.

Both insertions and deletions are performed in $O(\log_B N)$ I/Os, since the changes needed on the

leaf level can be performed in $O(1)$ I/Os, and the "rebalancing" at most propagates from l to the root of the current version. Since $\Omega(\gamma B)$ updates have to be performed in a leaf from the time it is created until a new rebalancing operation is needed, the total number of leaves created during N updates is $O(\frac{N}{\gamma B}) = O(\frac{N}{B})$. Each time a leaf is created or marked dead a corresponding insertion or deletion is performed recursively one level up the tree. Thus the number of nodes created one level above a leaf during N updates is $O(\frac{N}{B^2})$. In total the number of nodes created is $\sum_{i=1}^{\log_B N} O(\frac{N}{B^i}) = O(\frac{N}{B})$. Details can be found in [11, 26].

Theorem 1 ([11, 26]) *After N insertions and deletions of elements from a total order into an initially empty persistent B-tree, the structure uses $O(N/B)$ space and supports range queries in any version in $O(\log_B N + T/B)$ I/Os. An update can be performed on the newest version in $O(\log_B N)$ I/Os.*

2.2 Modified persistent B-tree

In the persistent B-tree as described above, elements were stored in the leaves only. To search efficiently, internal nodes also contain elements ("routing elements"). In our discussion of the persistent B-tree so far we did not discuss precisely how these elements are obtained, that is, we did not discuss what element is inserted in *parent*(v) when a new node (or leaf) v is created and a new reference is inserted in *parent*(v). As in normal B-trees, a *copy* of the maximal element in v is used as a routing element in *parent*(v) when v is created. Since copies of an element can be present as routing elements in internal nodes long after the element is deleted, all elements stored in the persistent B-tree during its entire lifespan need to be comparable (the elements need to be totally ordered). As mentioned, this means that the persistent B-tree cannot be used to design an I/O-efficient vertical ray-shooting structure.

To obtain a persistent B-tree structure that only requires elements present in the same version to be comparable, we modify the structure to store actual elements in all nodes; we impose the new invariant that the alive elements at any time t form a B-tree with data elements in internal as well as leaf nodes (as opposed to just in leaves). Except for slight modifications to the version-split, split, merge, and share operations, the insert algorithm remains unchanged. In the delete algorithm we need to be careful when deleting an element x in an internal node u. Since x is associated with a reference to a child u_c of u, we cannot simply mark x dead by updating its existence interval. Instead, we first find its predecessor y in a leaf below u and persistently delete it. Then we delete x from u, insert y with a reference to the child u_c, and perform the relevant rebalancing. What remains is to describe the modifications to the rebalance operations.

Version-split. Recall that a version-split (not leading to a strong underflow or overflow) consists of copying all alive elements in a node u, using them to create a new node v, deleting the reference to u and recursively inserting a reference to v in *parent*(u). Since the reference to u has an element x associated with it, we cannot simply mark it deleted by updating its existence interval. However, since we are also inserting a reference to the new node v, and since the elements in v are a subset of the elements in u, we can use x as the element associated with the reference to v. Thus we can perform the version-split almost exactly as before, while maintaining the new invariant, by simply using x as the element associated with the reference to v as shown in Figure 3.

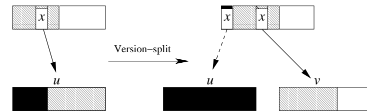


Figure 3: Illustration of a version-split. Partially shaded regions corresponds to alive elements, black to dead elements, and white regions to unused space. (The same shading is used at the top of individual elements to indicate their status).

Split. When a strong overflow occurs after a version-split of u, a split is needed; two new nodes v and v' are created and two references need to be inserted in *parent*(u). As in the version-split case, the element x associated with u in *parent*(u) can be used as the element associated with the reference to v'. To maintain the new invariant we then "promote" the maximal element y in v to be used as the element associated with the reference to v in *parent*(u), that is, instead of storing y in v, we store it in *parent*(u). Otherwise a split remains unchanged. Refer to Figure 4.

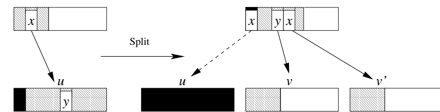


Figure 4: Illustration of a split.

Merge. When a strong underflow occurs after a version-split of u, we perform a version-split of u's sibling u' and create a new node v with the obtained elements. We then delete the references to u and u' in *parent*(u) by marking the two elements x and y

associated with the references to u and u' as deleted. We can reuse the maximal of these elements, say y, as the reference to the new node v. To maintain the new invariant (preserve all elements) we then "demote" x and store it in the new node v. Otherwise a merge remains unchanged. Refer to Figure 5.

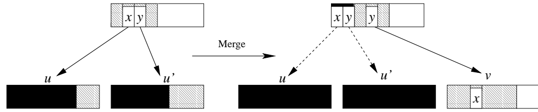


Figure 5: Illustration of a merge.

Share. When a merge would result in a new node with a strong overflow, we instead perform a share. We first perform a version-split on the two sibling nodes u and u', and create two new nodes v and v' with the obtained elements. As in the merge case, we delete the references to u and u' in $parent(u)$ by marking two elements x and y associated with u and u' as deleted. We can reuse the maximal element y as the reference to v' but x cannot be used as a reference to v. Instead, we demote x to v and promote the maximal element z in v to $parent(u)$. Refer to Figure 6.

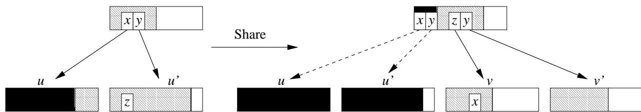


Figure 6: Illustration of a share.

Theorem 2 *After N updates on an initially empty modified persistent B-tree, the structure uses $O(N/B)$ space and supports range queries in any version in $O(\log_B N + T/B)$ I/Os. An update can be performed on the newest version in $O(\log_B N)$ I/Os.*

Corollary 1 *A set of N non-intersecting segments in the plane can be processed into a data structure of size $O(N/B)$ in $O(N \log_B N)$ I/Os such that a vertical ray-shooting query can be answered in $O(\log_B N)$ I/Os.*

While trivially performing a sequence of N given updates on a (modified as well as unmodified) persistent B-tree takes $O(N \log_B N)$ I/Os, it has been shown how the N updates can be performed in $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os (the sorting bound) on a normal (unmodified) persistent B-tree [25, 3, 6]. In the modified B-tree case, the lack of a total order seems to prevent us from performing the updates in this much smaller number of I/Os. In fact, the existence of such a fast algorithm would immediately lead to a semi-dynamic insert-efficient vertical ray-shooting structure using the external version of the logarithmic method [8].

3 Experimental results

In this section we describe the results of an extensive set of experiments designed to evaluate the performance of the (modified) persistent B-tree when used to answer vertical ray-shooting queries, compared to the grid structure of Vahrenhold and Hinrichs [24].

3.1 Implementations

We implemented both the persistent B-tree and the grid structure using TPIE. TPIE is a library of templated `C++` classes and functions designed to ease the implementation of I/O-efficient algorithms and data structures [5, 7]. Below we discuss the two implementations separately.

Persistent B-tree implementation. When using the persistent B-tree to answer vertical ray-shooting queries the elements (segments) already implicitly contain their existence interval (the x-coordinates of the endpoints). Thus we implemented the structure without explicitly storing existence intervals.[2] This way each element occupied 28 bytes. To implement the root B-tree we used the standard B-tree implementation in the TPIE distribution [7]. In this implementation each root element requires 16 bytes.

Two parameters α and γ are used in the definition of the persistent B-tree. Since we are working with large datasets, we choose these parameters to optimize for space. Space is minimized when γ (and thus the number of updates needed between creation of nodes) is maximized, and the constraints on α and γ require that we choose $\gamma < \min\{\alpha - 1/B, 1/3 - 2\alpha + 1/B\}$. The maximum value of γ is when $\alpha = 1/5 + \frac{4}{5B}$, so we chose $\alpha = 1/5$ and $\gamma = \frac{1}{5} - \frac{1}{B}$ accordingly. If we wanted to optimize for query performance, we should choose a larger value of α—leading to a smaller tree height—but a few initial experiments indicated that increasing α had relatively little impact on query performance.

Grid structure implementation. The idea in the grid structure of Vahrenhold and Hinrichs [24] is to impose a grid on the (minimal bounding box of the) segments and store each segment in a bucket corresponding to each grid cell it intersects. The grid is designed to have N/B cells and ideally each bucket is of size B. To adapt to the distribution of segments, a slightly different grid than the natural $\sqrt{N/B} \times \sqrt{N/B}$ grid is used; two parameters, $F_x = \frac{1}{N \cdot x_d} \sum_{i=1}^{N} |x_{i2} - x_{i1}|$ and $F_y = \frac{1}{N \cdot y_d} \sum_{i=1}^{N} |y_{i2} - y_{i1}|$, where the i'th segment is given by $((x_{i1}, y_{i1}), (x_{i2}, y_{i2}))$, are used to estimate the

[2] We have also implemented a general persistent B-tree with existence intervals. Both implementations will be made available in the next TPIE release.

amount of overlap of the x-projections and y-projections of the segments, respectively. Then the number of rows and columns in the grid is calculated as $N_x = \alpha_x \sqrt{N/B}$ and $N_y = \alpha_y \sqrt{N/B}$, where $\alpha_x/\alpha_y = F_y/F_x$ and $\alpha_x \alpha_y = 1$.

In the TPIE implementation of the grid structure [24], the segments are first scanned to compute N_x and N_y. Then they are scanned again and a copy of each segment is made for each cell it crosses. Each segment copy is also augmented with a bucket ID, such that each copy occupies 32 bytes. Finally, the new set of segment is sorted by bucket ID using TPIE's I/O-optimal merge sort algorithm [2]. The buckets are stored sequentially on disk and a *bucket index* is constructed in order to be able to locate the position of the segments in a given bucket efficiently. The index is simply a $N_x \times N_y$ array with entry (i, j) containing the position of the first segment in the bucket corresponding to cell (i, j) (as well as the number of segments in the bucket). Each index entry is 16 bytes.

If L is the the number of segment copies produced during the grid structure construction, $O(\frac{L}{B} \log_{M/B} \frac{L}{B})$ I/Os is the number of I/Os used by the algorithm. Ideally, this would be $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/Os, but in the worst case, each segment can cross $\sqrt{N/B}$ buckets and the algorithm requires $O(\frac{N}{B} \sqrt{N/B} \log_{M/B} \frac{N}{B})$ I/Os. Refer to Figure 7 for an example of such a dataset.

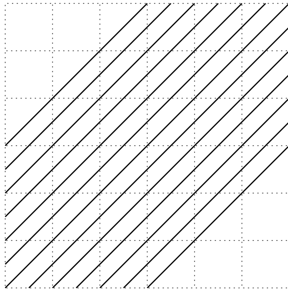


Figure 7: Worst-case dataset for the grid method.

Answering a query using the grid method simply involves looking up the position of the relevant bucket (the cell containing the query point) using the bucket index and then scanning the segments in the bucket to answer the query. In the ideal case, each bucket contains $O(B)$ segments and the query can be performed in $O(1)$ I/Os. However, in the worst case, a query takes $O(N/B)$ I/Os.

3.2 Data

To investigate the efficiency of our vertical ray-shooting data structure, we used road data from the US Census TIGER/Line dataset containing all roads in the United States [22]. In this dataset one curved road is represented as a series of short segments. Roads (segments) are also broken at intersections, such that no two segments intersect other than at endpoints. Because of various errors, the dataset actually contains a few intersection segments, which we removed in a preprocessing step.

Our first 6 datasets, containing between 16 million segments (374 MB) and 80 million segments (1852 MB), consist of the roads in connected parts of the US (corresponding to the six CD's on which the data is distributed). A summary of the number of segments in each dataset is shown in Table 1, and pictures of the datasets appear in Figure 8. In addition to these large datasets, we also used four smaller datasets with disjoint data regions; dataset CL1 consists of the four US states Washington, Minnesota, Maine and Florida, and CL2 excludes Minnesota. The bounding boxes of CL1 and CL2 are relatively empty with the exception of three or four "hot spots" where the states are located. These datasets were included to investigate the effect of non-uniform data distributions. The dataset DST spans a sparsely populated region of the United States and was included to investigate the effect of longer but more uniformly distributed segments; we expect that the segments are longer and more uniformly distributed in the desert than in metropolitan areas. The dataset ATL, on the other hand, was chosen as a dense, but

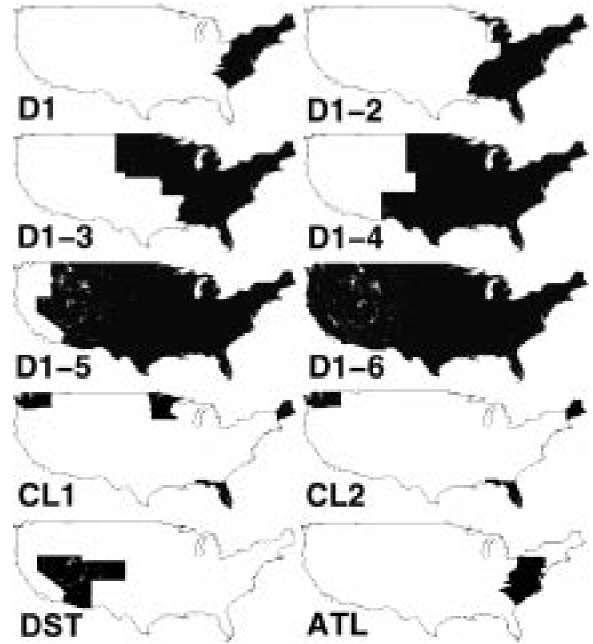


Figure 8: Illustration of the real-world datasets used in our experiments. Black regions indicate areas in the dataset. The outline of the continental US serves as a visualization guide and is not part of the actual data set.

Data Set	Segments (in Millions)	Size (MB)
D1	16.36	374
D1-2	31.22	714
D1-3	41.78	956
D1-4	57.33	1312
D1-5	69.82	1598
D1-6	80.91	1852
CL1	6.69	153
CL2	5.09	116
ATL	10.84	248
DST	6.40	146
LONG	1.00	23

Table 1: Number of segments and raw dataset size (assuming 24 bytes per segment) of the datasets.

maybe less uniform, dataset. Finally, in addition to the real world data, we also generated a dataset LONG, designed to illustrate the worst case behavior of the grid method corresponding to Figure 7.

For query point sets, we generated a list of 100000 randomly sampled points from inside each of the datasets; for each query point p, we require that segments are hit by vertical rays emanating from p in both the positive and negative y-directions.

3.3 Experimental setup

We ran our experiments on an Intel PIII - 500MHz machine running FreeBSD 4.5. All code and data resided on a local 10000 RPM 36GB SCSI disk. The disk block size was 8KB, so each leaf of the persistent B-tree could store 290 elements and each internal node (also containing references) 225 elements. In the root B-tree, each leaf could hold 510 elements and each internal node 681 elements. The grid method bucket index has $N/256$ entries of 16 bytes each. We limited the RAM of the machine to 128 MB and, since the OS was observed to use 60 MB, we limited the amount of memory available to TPIE to 12 MB. Constraining the memory provides insight into how the structures would perform if the system had more total memory but was under heavy load.

To increase realism in our experiments, we used TPIE's built-in (8-way set associative LRU-style) cache mechanism to cache parts of the data structures. Since a query on the persistent B-tree always begins with a search in the relatively small root B-tree, we used a 72 block cache (8 internal nodes, 64 leaf nodes) for the root B-tree—enough to cache the entire structure in all our experiments. We also used a separate 16 block cache for the internal persistent B-tree nodes and a 32 block cache for the leaf nodes. Separate caches were used for internal nodes and leaves to ensure that accesses to the many leaves did not result in eviction of the few internal nodes from the cache. In total the caches used for the entire persistent structure were of size 120 blocks or 960KB. For the grid structure, we (in analogy with the root B-tree in the persistent case) cached the entire bucket index—of size 2.41 MB for the largest dataset, D1-6.

3.4 Experimental results

Structure size. Figure 9 shows the size of the grid and persistent B-tree data structures constructed on the 11 datasets. For the real life (TIGER) data, the grid method uses about 1.4 times the space of the raw data, whereas the persistent B-tree sometimes uses almost 3 times the raw data size. The low overhead of the grid method is a result of relatively low segment duplication (around 1.02 average copies per segment) due to the very short segments. The rest of the space is used to store the bucket index. The larger overhead of the persistent B-tree is mainly due to each structural change (rebalance operation) resulting in the creation of multiple copies of each segment; we found that there are roughly 2.4 copies of each segment in a persistent B-tree structure.

Analyzing the numbers for the real datasets in more detail reveals that for the first six datasets and ATL the space utilization is quite uniform. For datasets DST, CL1, and CL2 the grid structure uses slightly less space while the persistent B-tree uses more space. The persistent B-tree method uses more space because the relatively small datasets are sparsely populated with segments. At any given time during the construction sweep, the sweep line intersects relatively few segments. As a result, many transitions are made between a height one (one leaf) and height two (with a low-degree root) tree, resulting in relatively low block utilization.

For the artificially generated dataset LONG, the space usage of both structures increases dramatically.

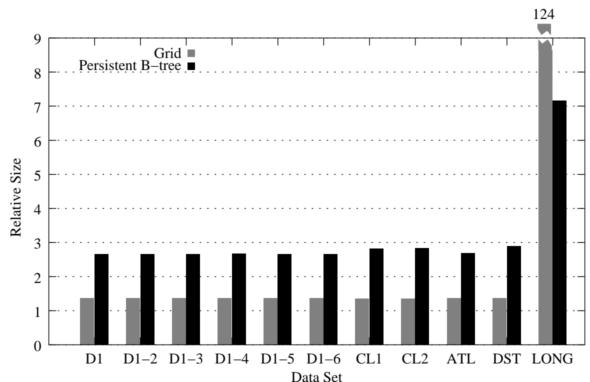


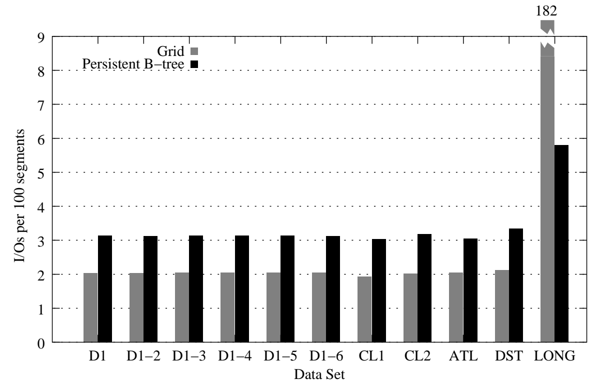
Figure 9: Space utilization of grid and persistent B-tree based structures. Size is relative to the raw dataset size.

As expected, the cause of the enormous space use of the grid structure is a high number of segment copies (93 per segment on the average). The persistent B-tree also has a significant increase in space, but not nearly as much as the grid structure—the structure is 94% smaller than the grid structure. The reason for increased space usage in the persistent B-tree is that all the segments are long and thus they stay in the persistent structure for a long time (in many versions), resulting in a high tree. Furthermore, most of the structural changes in the beginning of the construction algorithm are splits, leading to many copies of alive elements. Similarly, most of the structural changes at the end of the execution is merges, again leading to a large redundancy.

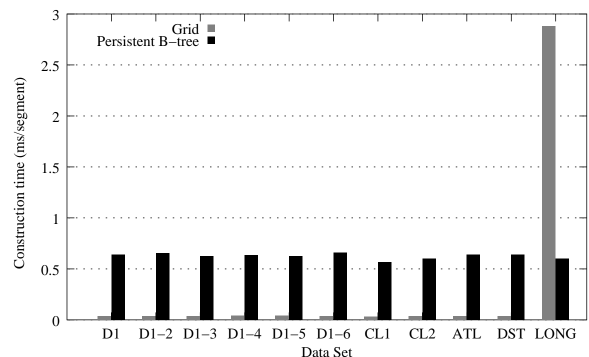
Construction efficiency. Figure 10 shows construction results in terms of I/O and physical time. For all the real life datasets, the persistent B-tree structure uses around 1.5 times more I/Os than the grid structure. This is rather surprising since the theoretical construction bound for the persistent B-tree is $O(N \log_B N)$ I/Os, compared to the (good case) $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ bound for the grid structure. Theory thus predicts that the tree construction algorithm should perform about B times as many I/Os as the grid method. The reason for this discrepancy between theory and practice is that during the construction sweep the average number of segments intersecting the sweep line is relatively small. For the D1-6 dataset, it is less than 2500 segments (see Figure 11). The size of the persistent B-tree accessed during each update is small and as a result the caches can store most of the nodes in the tree.

While it only takes about 50% more I/Os to construct the persistent B-tree structure than to construct the grid structure, it takes nearly 17 times more physical time. One reason for this is that most of the I/Os performed by the grid construction algorithm are sequential, while the I/Os performed by the persistent B-tree algorithm are more random. Thus the grid algorithm takes advantage of the optimization of the disk controller and the OS file system for sequential I/O, e.g., by prefetching. Another reason is that construction of the persistent B-tree structure is more computationally intensive than construction of the grid. Our trace logs show that over 95% of the construction time for the persistent B-tree is spent in internal memory compared to only 50% for the grid method.

While the grid construction algorithm outperforms the persistent B-tree algorithm on the real life datasets, the worst-case dataset, LONG, causes significant problems. For this dataset, the grid construction takes 48 minutes compared to 53 minutes for the 80 times bigger dataset D1-6, and compared to 10 minutes for the persistent B-tree. The reason is the large size of the structure results in a high I/O construction cost.



(a)



(b)

Figure 10: Construction performance: a) Number of I/Os per 100 segments. b) Construction time per segment.

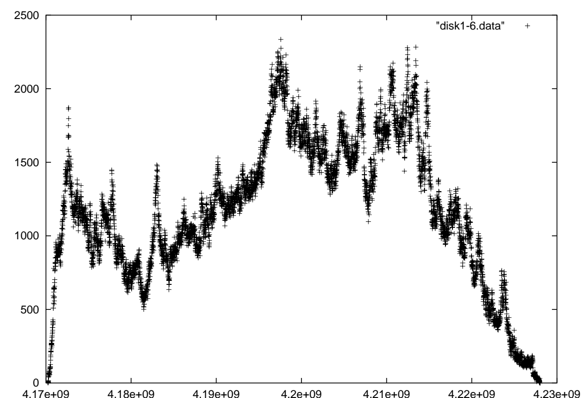
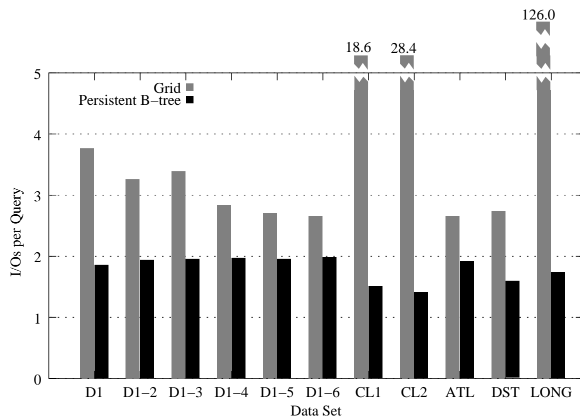


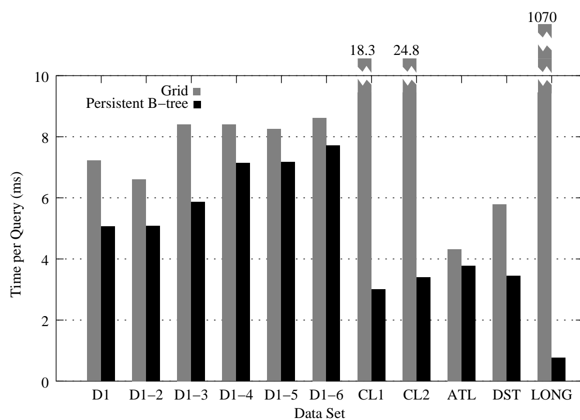
Figure 11: The number of segments intersecting the sweep line as a function of sweep line position during the construction of the persistent B-tree for the D1-6 dataset.

The persistent B-tree construction also takes relatively more I/Os (and time), mainly due to the high average number of segments intersecting the sweep-line (500000 at the peak).

Query efficiency. Our query experiments illustrate the advantage of the persistent B-tree structure over the grid structure; Figure 12 shows that a query is consistently answered in less than two I/Os on the average, while the grid structure uses between approximately 2.6 and 28 I/Os on the average for the real-world datasets, and 126 I/Os for the LONG dataset.



(a)



(b)

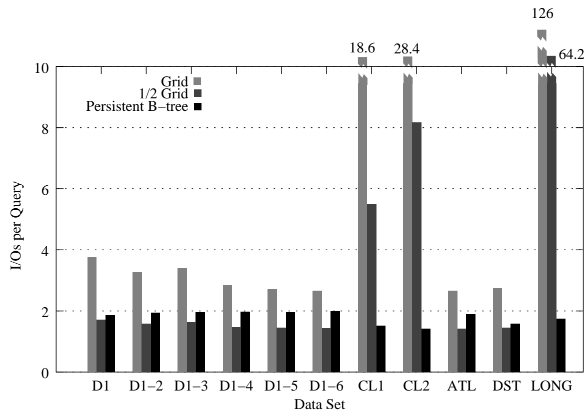
Figure 12: Query performance: a) Number of I/Os per query. b) Time per query in milliseconds.

Analyzing the grid structure results for the real-world datasets in more detail reveals that the performance is mostly a function of the distribution of segments within their bounding box. The bounding boxes become more full as we move from D1 to D1-6, and as a result the average number of I/Os per query drops from 3.76 to 2.65. The dataset ATL overlaps with a significant portion of D1, but because of the better distribution of segments within the bounding box the I/O performance is better for ATL. Datasets CL1 and CL2 exacerbate the problem with non-uniform distributions. For these datasets, most grid cells, more than 90%, are completely empty. As a result, a query within a non-empty cell is very expensive. Finally, as expected the LONG data set shows the grid structures vulnerability to long segments; on the average, a query takes 126 I/Os.

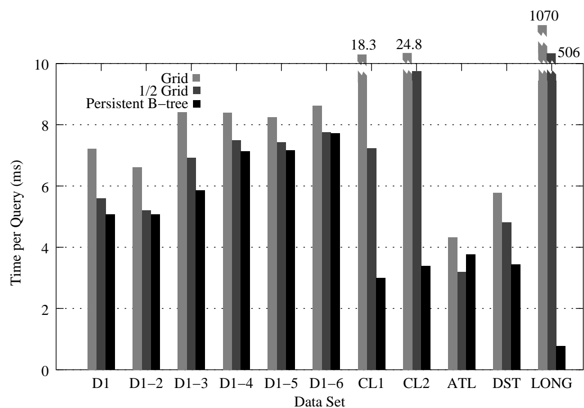
Analyzing the persistent B-tree results in detail reveals that its performance is mostly a function of the average number of segments intersecting the sweepline; datasets D1 through D1-6 and ATL have higher average I/O cost than DST, CL1 and CL2. Similarly, dataset D1 has a lower average cost than ATL, since the region of D1 not in ATL has a smaller number of sweepline-segment intersections. This is the opposite of the behavior of the grid method whose query performance is worse for D1 than ATL. Finally, even though the average number of sweepline-segment intersections for the LONG dataset is more than half a million, the height of the tree is no more than three at any time (version), and as a result the query efficiency is maintained.

Further experiments. In our experiments with the TIGER data, we noticed that non-empty buckets in the grid structure often contained three or more disk blocks of segments. Therefore we expected that by reducing the grid spacing in both the x and y-direction by a factor of two—creating four times as many buckets—each bucket would likely contain only one disk block, leading to an improved query performance. Such an improvement is of course highly data dependent. It would also come at the cost of space, since we must index four times as many buckets, and the denser grid may result in more segment duplication. To investigate this we ran our tests again using such a modified grid and found that for the real life datasets, the number of segment copies did not increase significantly (from 1.02 to 1.04 copies per segment). Thus the construction performance was maintained. In terms of query performance, we found that the four-fold increase in the number of buckets leads to a factor of two to three improvement in the query performance. Refer to Figure 13. In these experiments we cached the entire bucket index, which is four times larger than in the regular grid method. For D1 the index is 2 MB, and for D1-6 the index is 10 MB, ten times larger than the cache used in the persistent B-tree. For the LONG dataset, the modified grid uses twice the space of the standard grid and still has poor query performance compared to the persistent B-tree.

Finally, to investigate the influence of caches we ran a series of experiments without caching. In these



(a)



(b)

Figure 13: Query performance of the grid method using four times as many buckets (1/2 Grid): a) Number of I/Os per query. b) Time per query in milliseconds.

experiments we found that one additional I/O is used per query in the grid structure in order to access the bucket index. In the persistent B-tree structure one or two extra I/Os are used depending on the height of the root B-tree. This can immediately be reduced by one I/O per query by just caching the block containing the root of the root B-tree. Thus we conclude that the query performance does not depend critically on the existence of caches.

4 Conclusions

In this paper, we have presented an external point location data structure based on a persistent B-tree that is efficient both in theory and practice. One major open problem is to construct the structure in $O(\frac{N}{B}\log_{M/B}\frac{N}{B})$ I/Os, as compared to the (trivial) $O(N\log_B N)$ algorithm discussed in this paper.

Acknowledgments

We thank Jan Vahrenhold for providing and explaining the grid method code and Tavi Procopiuc for help with the TPIE implementation.

References

[1] P. K. Agarwal, L. Arge, G. S. Brodal, and J. S. Vitter. I/O-efficient dynamic point location in monotone planar subdivisions. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 1116–1127, 1999.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

[4] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer Academic Publishers, 2002.

[5] L. Arge, R. Barve, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickremesinghe. *TPIE User Manual and Reference (edition 0.9.01a)*. Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.

[6] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. *Algorithmica*, 33(1):104–128, 2002.

[7] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. Annual European Symposium on Algorithms*, 2002.

[8] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. In *Proc. ACM Symp. on Computational Geometry*, pages 191–200, 2000.

[9] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995. To appear in special issues of *Algorithmica* on Geographical Information Systems.

[10] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[11] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB Journal*, 5(4):264–275, 1996.

[12] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[13] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. *International Journal of Computational Geometry & Applications*, 11(3):305–337, June 2001.

[14] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[15] M. Edahiro, I. Kokubo, and T. Asano. A new point-location algorithm and its practical efficiency — Comparison with existing algorithms. *ACM Trans. Graph.*, 3(2):86–109, 1984.

[16] H. Edelsbrunner and H. A. Maurer. A space-optimal solution of general region location. *Theoret. Comput. Sci.*, 16:329–336, 1981.

[17] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[18] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[19] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.

[20] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669–679, 1986.

[21] J. Snoeyink. Point location. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press LLC, Boca Raton, FL, 1997.

[22] *TIGER/Line*™ *Files, 1997 Technical Documentation*. Washington, DC, September 1998. http://www.census.gov/geo/tiger/TIGER97D.pdf.

[23] U.S. Geological Survey. 1:100,000-scale line graphs (DLG). Accessible via URL. `http://edcwww.cr.usgs.gov/doc/edchome/ndcdb_bk/` (Accessed 9 September 2002).

[24] J. Vahrenhold and K. H. Hinrichs. Planar point location for large data sets: To seek or not to seek. In *Proc. Workshop on Algorithm Engineering, LNCS 1982*, pages 184–194, 2001.

[25] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proc. International Conf. on Very Large Databases*, pages 406–415, 1997.

[26] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, 1997.