Lecture Notes: External Priority Search Tree

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

In this lecture, we will consider another fundamental problem in computer science: *3-sided range searching*. Let P be a set of N points in $\mathbb{R}^2$. A rectangle is said to be *3-sided* if it has the form $[x_1, x_2] \times [y, \infty)$, namely, its bottom edge is grounded at the bottom of the data space. Given a 3-sided rectangle q, a *3-sided range query* reports all the points of P covered by q, namely, $P \cap q$. This problem generalizes the stabbing problem we discussed previously (think: why?).

Interestingly, the persistent B-tree can also be used to solve the static version of the problem.

Lemma 1. *There is a persistent B-tree that consumes $O(N/B)$ space and answers a 3-sided range query in $O(\log_B N + K/B)$ I/Os, where K is the number of points reported.*

Proof. From each point $p \in P$, create a vertical ray shooting downwards from p. Let R be the set of all such rays. Then, p falls in a 3-sided rectangle $q = [x_1, x_2] \times [y, \infty)$ if and only if its ray intersects the horizontal segment $[x_1, x_2] \times y$. Hence, we can instead find all the rays in R intersecting $[x_1, x_2] \times y$, a problem that can be solved by a persistent B-tree with the performance claimed. □

Next, we will introduce the *external priority search tree* [1], which is a dynamic structure that has the same space and query cost as the persistent B-tree, but also supports an insertion and a deletion in $O(\log_B N)$ I/Os. Our discussion will make the tall cache assumption $M \geq B^2$. We also assume that P is in *general position*, namely, no two points in P have the same x- or y-coordinate.

1 Structure

The base tree of the external priority search tree is a weight balanced B-tree $\mathcal{T}$ on the set of x-coordinates of the points in P. The leaf and branching parameters of $\mathcal{T}$ are both set to B. Each node u in $\mathcal{T}$ naturally corresponds to a vertical *slab* $\sigma(u)$ in $\mathbb{R}^2$. Denote by $sub(u)$ the subtree of u.

Each node u is associated with a *pilot set* denoted as $pilot(u)$. Next, we define the pilot sets in a top-down fashion:

- Let v_{root} be the root of $\mathcal{T}$. If v_{root} is a leaf, then $pilot(v_{root})$ is simply P itself. Otherwise, suppose that v_{root} has f child nodes $u_1, ..., u_f$. Then, $pilot(v_{root})$ is the union of the B highest points from each $sub(u_i)$, for $i \in [1, f]$.

- Now consider an internal node v with f child nodes $u_1, ..., u_f$. Let $pilot(v, u_i)$ be the B highest points in $sub(u_i)$ after *excluding* the points that already appear in the pilot sets of the proper ancestors of v. If less than B points satisfy the condition, $pilot(v, u_i)$ includes all of them. Then, the pilot set $pilot(v)$ of v simply unions $pilot(v, u_1)$, ..., $pilot(v, u_f)$.

- Finally, for a leaf node z, $pilot(z)$ is the set of points in $\sigma(z)$ that do not belong to the pilot set of any proper ancestor of z.

Note that each pilot set has at most B^2 points.

For each internal node v, we associate v with a persistent B-tree $T(v)$ built on $pilot(v)$. To facilitate updates, we use a B-tree $T'(u)$ to index the y-coordinates of the points in $pilot(u)$. If z is a leaf node, it is associated with just an extra block to store $pilot(z)$. The overall space consumption is $O(N/B)$ (think: why?).

2 Query

We answer a query by reporting points *only* from the pilot sets. Given a query rectangle $q = [x_1, x_2] \times [y, \infty)$, descend a root-to-leaf path Π_1 (Π_2) to the leaf node whose slab contains x_1 (x_2). For each node $u \in \Pi_1 \cup \Pi_2$, launch the following *filtering search* process:

- If u is a leaf node, simply report all the points in $pilot(u)$ covered by q.
- Otherwise, suppose that $u_{i_1}, ..., u_{i_2}$ are the child nodes of u such that σ_j ($i_1 \le j \le i_2$) is contained in $[x_1, x_2] \times \mathbb{R}$. Let $q' = \sigma_{i_1} \cup \sigma_{i_1+1} \cup ... \cup \sigma_{i_2}$. Search $T(u)$ to report all the points in $pilot(u)$ covered by q'. For each $j \in [i_1, i_2]$ such that B points have been reported, perform the filtering search process on u_j.

The above algorithm correctly finds all the points in $P \cap q$ (think: why?).

For each node u visited by the query algorithm, we spend $O(1 + K_u/B)$ I/Os (see Lemma 1), where K_u is the number of points reported from $T(u)$. Refer to the term "1" as the *search cost* at u. The nodes visited can be divided into two groups: (i) those on Π_1 and Π_2, and (ii) those that are not (note that any such node u must have its slab $\sigma(u)$ covered completely by $[x_1, x_2] \times \mathbb{R}$). For each node u of the second group, $\Omega(B)$ points in $\sigma(u)$ must have been reported at the parent of u. Hence, we charge the search cost of u on those points. In this way, each point reported bears $O(1/B)$ additional I/Os. The overall query cost is therefore $O(\log_B N + K/B)$ (think: how to account for the nodes of the first group?).

3 Updates

Next, we will make the external priority search tree dynamic.

3.1 The B^2-Structure

Recall that each node u is associated with a persistent B-tree $T(u)$. By applying the "single buffer block" trick for $T(u)$ (see Lemma 2 of the lecture nodes on the external interval tree), we have:

Lemma 2. *Under the tall-cache assumption, $T(u)$ can be updated in $O(1)$ amortized I/Os per insertion and deletion.*

3.2 Demotion

Given a point p and a node u such that $p \in \sigma(u)$, a *demotion* operation adds p to the unique pilot set (of some node) in $sub(u)$ that should contain p, according to the pilot set definition. If u is a leaf node, we simply place p in the block storing $pilot(u)$.

Now consider that u is an internal node. Let u' be the child node of u such that $\sigma(u')$ contains p. If $pilot(u, u')$ currently has less than B points, we finish by adding p to $pilot(u)$, updating $T(u)$ and $T'(u)$ accordingly. Otherwise, we use $T(u)$ to find the lowest point, say p', in $pilot(u, u')$ in $O(1)$ I/Os (think: how?). Then:

- If p is higher than p', remove p' from $pilot(u)$ and add p to $pilot(u)$ by updating $T(u)$ and $T'(u)$ appropriately. After this, perform a demotion operation with p' and u'.
- Otherwise, simply perform a demotion operation with p and u'.

In general, if u is at level l, in the worst case we perform constant I/Os at each node along a single path from u to a leaf node. Hence, a demotion finishes in $O(l + 1)$ I/Os.

3.3 Promotion

Conversely, given a node u, sometimes we need to perform a *promotion* operation to remove from $pilot(u)$ the highest point p there, if $pilot(u)$ is not empty. If u is a leaf node, this is trivial.

Now consider that u is an internal node. We first obtain p from $T'(u)$ in $O(1)$ I/Os. Then, we remove p from $pilot(u)$, updating $T(u)$ and $T'(u)$ appropriately. Suppose that u' is the child node of u whose slab $\sigma(u')$ contains p. Recursively promote a point, say p', from $pilot(u')$, and add p' to $pilot(u)$, updating $T(u)$ and $T'(u)$ appropriately.

In general, if u is at level l, the promotion takes $O(l + 1)$ I/Os.

3.4 Insertion

Assume that p is the point being inserted. We first insert the x-coordinate of p in $\mathcal{T}$, without handling the overflows that may have happened. Let Π be the root-to-leaf path we just followed. Launch a demotion operation with p and the root of $\mathcal{T}$. The cost so far is $O(\log_B N)$.

Now we handle in bottom-up order the nodes that have overflown during the insertion of p in $\mathcal{T}$. Let u be such a node and v its parent node. Split u into u_1, u_2 (as in the weight-balanced B-tree). Rebuild the secondary structures of u_1 and u_2 respectively in $O(B)$ I/Os (recall that each secondary structure indexes at most B^2 points, which fit in memory). The split has divided $pilot(v, u)$ into $pilot(v, u_1)$ and $pilot(v, u_2)$. Now $pilot(v, u_1)$ may have less than B points. Hence, we perform up to B promotions to fill up $pilot(v, u_1)$. Repeat the same for $pilot(v, u_2)$. After this, rebuild the secondary structures of v in $O(B)$ I/Os.

Assume that u is at level l. If $l = 0$, the overflow handling finishes in constant I/Os. Otherwise, the cost is $O(lB)$. As $\mathcal{T}$ is a weight-balanced B-tree, the weight of u is $\Theta(B^{l+1})$, meaning that $\Omega(B^{l+1})$ updates have been performed in $sub(u)$ since the creation of u. Hence, we can amortize the overflow handling cost over those updates, such that each of them bears $O(lB/B^{l+1}) = O(1)$. As each update can bear such a cost at most $O(\log_B N)$ times, each insertion can be performed in $O(\log_B N)$ I/Os amortized.

3.5 Deletion

It is easy to maintain the pilot sets in $O(\log_B N)$ I/Os per deletion (we leave the details to you but obviously you need to use promotion). Recall that, in answering a query, we report points only from pilot sets. This suggests that we can avoid underflows in the base tree $\mathcal{T}$ with global rebuilding, in a way similar to what we did in the external interval tree. With this, we conclude:

Theorem 1. *Under the tall-cached assumption, there exists a structure on a set of N points that uses $O(N/B)$ space, answers a 3-sided range query in $O(\log_B N + K/B)$, and can be updated in $O(\log_B N)$ amortized I/Os per insertion and deletion.*

Remarks. Arge, Samoladas and Vitter [1] showed that the above theorem still holds even without the tall-cache assumption, and that the update cost can be made worst-case. The filtering search idea was first proposed by Chazelle [2].

References

[1] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *PODS*, pages 346–357, 1999.

[2] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. of Comp.*, 15(3):703–724, 1986.