Lecture Notes: External Interval Tree

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

This lecture discusses the *stabbing problem.* Let $\mathcal{I}$ be a set of N intervals in $\mathbb{R}$. We want to store $\mathcal{I}$ in a structure so that, given a query value $q \in \mathbb{R}$, all the intervals in $\mathcal{I}$ containing q can be reported efficiently. Such a query is called a *stabbing query.*

This problem can actually be solved by the persistent B-tree:

Lemma 1. *We can pre-process a set of N intervals into a persistent B-tree so that a stabbing query can be answered in $O(\log_B N + K/B)$ I/Os, where K is the number of intervals reported.*

We leave the proof to you. Both the space and query complexities are known to be optimal (the optimality proof requires knowledge about some sophisticated lower bounds in the cell probe model, and is beyond the scope of this course).

We will focus on the *dynamic* setting of the problem, where the structure needs to support insertions and deletions, and yet, still answer queries efficiently. The persistent B-tree falls short for this purpose. We will discuss an alternative structure called the *external interval tree* designed by Arge and Vitter [1]. The structure uses linear space, answers a query in $O(\log_B N + K/B)$ I/Os, and supports an update (both insertion and deletion) in $O(\log_B N)$ I/Os.

For simplicity, we assume that the intervals in $\mathcal{I}$ are in general position, such that all the (left and right) endpoints in $\mathcal{I}$ are distinct. Also, we make the *tall cache assumption* that $M \geq B^2$.

1 External Interval Tree—The Static Version

Structure. The base tree of the external interval tree is a B-tree $\mathcal{T}$ on the set of $2N$ endpoints with leaf parameter $B \geq 16$ and branching parameter $\sqrt{B}$. Naturally, each node u in $\mathcal{T}$ corresponds to a range $\sigma(u)$—called the *slab* of u—in the form $[x_1, x_2)$. Specifically, if u is a leaf node, then x_1 is the smallest element stored in u, while x_2 is the smallest element stored in the leaf succeeding u (if such a leaf does not exist, $x_2 = \infty$). If, on the other hand, u is an internal node with child nodes[1] $u_1, u_2, ..., u_f$, then $\sigma(u)$ is the union of $\sigma(u_i)$ for all $i \in [1, f]$. Given i, j satisfying $1 \leq i \leq j \leq f$, we define a *multi-slab* $\sigma_u[i, j]$ of u as $\sigma(u_i) \cup \sigma(u_{i+1}) \cup ... \cup \sigma(u_j)$. There are $\sqrt{B} + (\sqrt{B} - 1) + ... + 1 < B$ multi-slabs.

We associate u with a *stabbing set* S_u of intervals: an interval $s \in \mathcal{I}$ is assigned to the stabbing set of the lowest node in $\mathcal{T}$ whose slab contains s.

Consider an internal node v with child nodes $u_1, ..., u_f$. The fact $s \in S_v$ implies that s is not covered by the slab of any child node of v. Thus, s can be cut into at most 3 pieces: s^l, s^m, s^r, referred to as the *left*, *middle*, and *right* interval of s, respectively. Specifically, let i (j) be the child slab $\sigma(u_i)$ (or $\sigma(u_j)$) that covers the left (or right, resp.) endpoint of s. Then $s^l = s \cap \sigma(u_i)$, $s^r = s \cap \sigma(u_j)$, and $s^m = s \setminus (s^l \cup s^r)$. Note that $s^m \neq \emptyset$ if and only if $j \geq i + 2$. Denote by S_v^l, S_v^m, S_v^r the set of left, middle, right intervals generated from S, respectively.

[1]We always list the child nodes of an internal node in ascending ordering, i.e., the leaf elements underneath u_i are smaller than those underneath of u_j for any $i < j$.

Notice that s^m is precisely the largest multi-slab of u spanned by s. We say that s^m *belongs* to the multi-slab, and that a multi-slab is *underflowing* if it has less than B intervals. As there are less than B multi-slabs, no more than B^2 intervals in total belong to the underflowing multi-slabs.

We associate v with several secondary structures:

- $L_v(i)$ for each $i \in [1, f]$: A linked list[2] storing in ascending order the left endpoints of the intervals in S_v^l covered by $\sigma(u_i)$. We refer to the set $\{L_v(1), ..., L_v(f)\}$ collectively as the *left structure* of v.

- $R_v(i)$ for each $i \in [1, f]$: A linked list storing in descending order the right endpoints of the intervals in S_v^r covered by $\sigma(u_i)$. Refer to $\{R_v(1), ..., R_v(f)\}$ collectively as the *right structure* of v.

- For each multi-slab $\sigma_v[i, j]$ that does *not* underflow, create a linked list $M_v[i, j]$ containing all the middle intervals of $\sigma_v[i, j]$. Refer to the set of all such linked lists collectively as the *middle structure* of v.

- A persistent B-tree U_v on the at most B^2 intervals of the underflowing multi-slabs. Refer to U_v as the *underflow structure* of v.

Consider a leaf node z. S_z has at most $B/2$ intervals, noticing that each interval in S_z must have *both* endpoints in $\sigma(z)$. Hence, we can store S_z in a single block.

Space. Each leaf node uses only constant blocks. An internal node v, on the other hand, requires $O(\sqrt{B} + |S_v|/B)$ blocks. Since (i) each interval of $\mathcal{I}$ is assigned to only one stabbing set, and (ii) there are $O(N/(B\sqrt{B}))$ internal nodes, the overall space consumption is $O(N/B)$.

Query. We answer a stabbing query with a search value q by reporting intervals only from stabbing sets. First, descend a root-to-leaf path Π of $\mathcal{T}$ to reach the leaf node whose slab contains q. For each internal node $v \in \Pi$, the intervals in S_v covering q can be divided into four categories:

1. Their left intervals cover q.

2. Their right intervals cover q.

3. Their middle intervals cover q and belong to multi-slabs that do not underflow.

4. Their middle intervals cover q and belong to underflowing multi-slabs.

Let $u_1, ..., u_f$ be the child nodes of u. Assume that q falls in $\sigma(u_i)$ for some i. To report the intervals of Category 1 (Category 2 is symmetric), we scan $L_v(i)$ in its sorted order and report the intervals seen until either having exhausted the list or encountering a left endpoint greater than q. The cost of doing so is $O(1 + K_1/B)$ where K_1 is the number of Category-1 intervals.

To report the intervals of Category 3, we simply output *all* the intervals in $M_v[i, j]$ for each multi-slab $\sigma_u[i, j]$ covering q that does not underflow. The cost is $O(K_3/B)$ I/Os, where K_3 is the number of Category-3 intervals.

Finally, we obtain the intervals of Category 4 by simply querying the persistent B-tree U_v, which, by Lemma 1, takes $O(1 + K_4/B)$ I/Os, where K_4 is the number of Category-4 intervals. Overall, at v, we spend $O(1 + K_v/B)$ I/Os, where K_v is the number of result intervals from S_v. Thus, we conclude that the total query cost is $O(\log_B N + K/B)$ I/Os (think: we have not discussed how to process the leaf node on Π, but this is trivial. Why?).

[2]A linked list in external memory is simply a chain of blocks each of which contains $\Omega(B)$ elements except possibly the last one.

Remarks. There are two crucial ideas behind the static version of the external interval tree. The first one is to set the branching parameter to $\sqrt{B}$. This ensures that there are B multi-slabs at each internal node, and that meanwhile the tree height remains $O(\log_B N)$. The second one is to use a structure of query time $O(\text{polylog}_B N)$ to manage B^2 intervals at an internal node. A query on such a "B^2-structure" incurs only $O(\text{polylog}_B B^2) = O(1)$ I/Os.

2 Making the External Interval Tree Dynamic

2.1 A Bruteforce Method

Let us start with a very simple trick that will be useful later. Consider, in general, a data structure T that manages *at most* N elements. Assume that (i) T occupies $\mathit{space}(N)$ blocks, (ii) can be constructed in $\mathit{build}(N)$ I/Os, and (iii) supports a query in $\mathit{query}(N) + O(K/B)$ I/Os, where K is the number of elements reported. Then, we have:

Lemma 2. *T can be converted into a dynamic structure that has size $\mathit{space}(N)$, answers a query in $O(\mathit{query}(N)+K/B)$ I/Os, and supports an insertion or a deletion in $\frac{1}{B}\cdot \mathit{build}(N)$ I/Os amortized.*

To achieve the above purpose, it suffices to associate T with one *buffer block* in the disk. Given an update, we simply place it in the buffer block without actually modifying T. This way, the space complexity of T remains $\mathit{space}(N)$. To answer a query, we first retrieve from T the set S of qualifying elements in $\mathit{query}(N) + O(|S|/B)$ I/Os. Remember, however, some elements in S may no longer belong to the dataset due to the deletions in the buffer block. Conversely, some new elements to be added to the dataset by the insertions in the buffer may also need to reported. To account for these changes, it suffices to spend an extra I/O to inspect the buffer block. In any case, $|S|$ cannot differ from K by more than B. Hence, the total query cost is $\mathit{query}(N) + 1 + O((K + B)/B) = O(\mathit{query}(N) + K/B)$.

How to incorporate the buffered updates into T? We do nothing until the buffer has accumulated B updates. At this time, simply rebuild the entire T in $\mathit{build}(N)$ I/Os, and then clear the buffer. Since the rebuilding happens once every B updates, on average, each update bears only $\mathit{build}(N)/B$ I/Os.

2.2 Modifying the Structure

We need to slightly modify the static external interval tree to support updates. The base tree $\mathcal{T}$ is now implemented as a weight-balanced B-tree with leaf parameter B and branching parameter $\sqrt{B}$. Consider an internal node v in $\mathcal{T}$ with f child nodes. Before, each $L_v(i)$ ($1 \le i \le f$) was implemented as a linked list, but now we implement it as a B-tree indexing the left endpoints therein. Similarly, $R_v(i)$ now also becomes a B-tree on the right endpoints of the intervals in the tree. Similarly, we implement each $M_v[i,j]$ as a B-tree indexing left endpoints.

Remember that each multi-stab $\sigma_v[i,j]$ of v has some middle intervals. We will stick to the invariant that *all* those intervals are stored in either the middle structure $M_v[i,j]$, or the underflow structure U_v. Previously, a multi-slab $\sigma_v[i,j]$ is "underflowing" if it has less than B middle intervals. Now, we need to redefine this notion:

- If the middle intervals of $\sigma_v[i,j]$ are in the underflow structure U_v, $\sigma_v[i,j]$ is underflowing if the number of those intervals is below B.

- Otherwise, $\sigma_v[i,j]$ is underflowing is the number of middle intervals is below $B/2$.

At all times, the middle intervals of $\sigma_v[i, j]$ are stored in U_v *if and only if* $\sigma_v[i, j]$ is underflowing (think: how do these intervals move between U_v and $M_v[i, j]$?).

The above changes do not affect the space consumption of the overall structure, and nor do they affect the query algorithm or its cost.

2.3 The Underflow Structure

Recall that each internal node v is associated with a persistent B-tree U_v, which manages at most B^2 intervals, and hence, uses $O(B)$ space. It answers a stabbing query on those intervals in $O(1+K/B)$ I/Os. Under the tall cache assumption, we can easily build it in $O(B)$ I/Os by reading all the at most B^2 intervals into memory, creating the structure in memory, and then writing it back to the disk. Applying Lemma 2, we have already made U_v dynamic:

Corollary 1. *Each underflow structure consumes $O(B)$ space, answers a stabbing query (on the indexed intervals) in $O(1 + K/B)$ I/Os, and supports an insertion or a deletion in $O(1)$ I/Os amortized.*

2.4 Insertion

Let s be the interval being inserted. We first insert the left and right endpoints of s in $\mathcal{T}$ (without handling node overflows even if they occur) by traversing at most two root-to-leaf paths. In doing so, we have also identified the node whose stabbing set should include s. If this is a leaf node z, we simply add s to its stabbing set S_z.

Updating the Stabbing Set of an Internal Node. Consider that s needs to be added to the stabbing set S_v of an internal node v. Assume that v has f child nodes $u_1, u_2, ..., u_f$, and that the left (or right) endpoint of s falls in $\sigma(u_i)$ (or $\sigma(u_j)$) for some i, j. Cut s into a left interval s^l, a middle interval s^m, and a right interval s^r. Insert s^l and s^r into $L_v(i)$ and $R_v(j)$, respectively.

Given a non-empty s^m, we check whether the middle intervals of $\sigma_v[i + 1, j - 1]$ are stored in $M_v[i + 1, j - 1]$. If so, s^m is inserted into $M_v[i + 1, j - 1]$ in $O(\log_B N)$ I/Os.

Otherwise, we add s^m to U_v, after which $\sigma_v[i + 1, j - 1]$ may have B middle intervals such that it no longer underflows. In this case, we retrieve all of them in $O(1)$ I/Os (by performing a stabbing query on U_v), delete them from U_v in $O(B)$ amortized I/Os (see Corollary 1), initialize a B-tree $M_v[i + 1, j - 1]$ with those B intervals in $O(1)$ I/Os. We can charge the $O(B)$ cost over the at least $B/2$ intervals that must have been added to U_v since the last movement of the middle intervals of $\sigma_u[i + 1, j - 1]$ from $M_v[i + 1, j - 1]$ to U_v. Therefore, on average, each insertion bears only $O(B)/\frac{B}{2} = O(1)$ I/Os for moving the B intervals from U_v to $M_v[i + 1, j - 1]$.

The cost so far is $O(\log_B N)$ amortized.

Overflow Handling. We proceed to handle the overflowing nodes (if any) on the (at most two) root-to-leaf paths we descended at the beginning. The overflows are treated in a bottom-up manner, namely, first handling the at most two leaf nodes, then their parents, and so on.

In general, let u be a node that overflows, and v be its parent. Split the elements of u into u_1 and u_2 as in the weight-balanced B-tree. Let ℓ be the splitting value, i.e., all the elements in u_1 (or u_2) are smaller (or at least, resp.) ℓ. Note that ℓ becomes a new slab boundary at v. We proceed to fix the secondary structures of u_1, u_2 and v.

The intervals in S_u (stabbing set of u) can be divided into three groups: those (i) completely to the left of ℓ, (ii) completely to the right of ℓ, and (iii) crossing ℓ. The first group becomes S_{u_1}, the second becomes S_{u_2}, while the intervals of the third group, denoted as S_{up}, should be inserted into S_v. S_{u_1}, S_{u_2}, S_{up} can be obtained in $O(|S_u|/B)$ I/Os by scanning S_u once. Also, in $O(|S_u|/B)$

I/Os, we can obtain two sorted lists for S_{u_1}, one sorted by the left endpoints of its intervals and the other by their right endpoints (the details are left to you). We refer to the first (or second) copy as the *left* (or *right*, resp.) copy of S_{u_1}. We obtain the two copies also for S_{u_2} and S_{up}, respectively, in $O(|S_u|/B)$ I/Os.

Lemma 3. *Let v be an internal node. Given the left and right copies of its stabbing set S_v, all the secondary structures of v can be built in $O(\sqrt{B} + |S_v|/B)$ I/Os.*

Proof. Assume that v has $f \le \sqrt{B}$ child nodes. By scanning the left copy of S_v once, we can generate the intervals indexed by $L_v(i)$ for each $i \in [1, f]$. After which, $L_v(i)$ can be built in $O(1 + |L_v(i)|/B)$ I/Os. Hence, the left structure of v can be constructed in $O(\sqrt{B} + |S_v|/B)$ I/Os in total. Similarly, its right structure can also be constructed from the right copy of S_v in the same cost.

As there are less than $f^2 = B$ multi-slabs and $M \ge B^2$, by scanning the left copy of S_v once, we can achieve two purposes:

- If a multi-slab has at least B middle intervals, all those intervals are stored in a file, sorted by left endpoint.
- Otherwise, all its middle intervals remain in the memory.

Build the underflow structure U_v on the intervals in memory, and write U_v to the disk in cost linear to the number of intervals in U_v. Finally, for each non-underflowing multi-slab $\sigma_v[i, j]$, build $M_v[i, j]$ in cost linear to the number of middle intervals of $\sigma_v[i, j]$. □

Therefore, given S_{u_1} and S_{u_2}, the secondary structures of u_1 and u_2 (if they are internal nodes) can be constructed in $O(\sqrt{B} + |S_{u_1}|/B + |S_{u_2}|/B) = O(B + |S_u|/B)$ I/Os. At v, the task is to union S_{up} into S_v. Given the left and right copies of S_{up}, it is easy to generate the left and right copies of S_v in $O(|S_v|/B)$ I/Os, after which the secondary structures of v can be rebuilt in $O(\sqrt{B} + |S_v|/B)$ I/Os.

We now show that the cost of overflow handling does not increase the amortized insertion cost. Remember that $\mathcal{T}$ is a weight-balanced B-tree with leaf parameter B and branching parameter $p = \sqrt{B}$. Let $w(u)$ and $w(v)$ be the weights of u and v, respectively. It thus follows that $w(v) \le 4p \cdot w(u)$. Observe that $|S_u| \le w(u)$ (as each interval in S_u has both endpoints in the subtree of u), and $|S_v| \le w(v)$. In other words, the total cost of re-constructing the secondary structures of u_1, u_2 and v is

$$\begin{aligned} O(\sqrt{B} + |S_u|/B + |S_v|/B) &= O(\sqrt{B} + w(u)/B + w(v)/B) \\ &= O(\sqrt{B} + w(v)/B) \\ &= O(\sqrt{B} + p \cdot w(u)/B) \\ &= O(w(u)) \end{aligned}$$

where the last equality used the fact that $w(u) \ge B\sqrt{B}$.

Recall that, as a property of the weight-balanced B-tree, when u overflows, $\Omega(u)$ updates have been performed under its subtree. Hence, we can amortize the $O(w(u))$ cost of overflow handing over those updates so that each of them accounts for only constant I/Os. As each update needs to bear such cost $O(\log_B N)$ times, it follows that each insertion is amortized only $O(\log_B N)$ I/Os.

2.5 Performing a Deletion

As shown above, the major difficulty in performing an insertion is the handling of node overflows. Thus, it is natural to expect that it would be equally difficult to handle node underflows in deletions. Interestingly, we can circumvent node underflows altogether by using a technique called *global rebuilding* [2]. Recall that a query algorithm reports intervals only from stabbing sets. Hence, as long as the stabbing sets are properly maintained, we are free to some "useless" elements in $\mathcal{T}$.

Specifically, to delete an interval s, we first remove it from the secondary structures of the node whose stabbing set contains s. This can be accomplished easily in $O(\log_B N)$ I/Os by reversing the corresponding steps in an insertion. We finish the deletion right here, in particular, *leaving* the left and right endpoints of s in $\mathcal{T}$. It is easy to see that the correctness of the query algorithm can still be guaranteed. As no element is ever deleted from $\mathcal{T}$, node underflows can never happen.

There is, however, a drawback. Since we permit useless endpoints to remain in $\mathcal{T}$, over time the number of endpoints in $\mathcal{T}$ can become so much larger than N, such that the height of $\mathcal{T}$ is no longer $O(\log_B N)$, which in turn compromises our query complexity. To remedy the drawback, we reconstruct $\mathcal{T}$ from time to time. Specifically, we create $\mathcal{T}$ for the first time when the size of the input set $\mathcal{I}$ has reached B (think: what to do before this moment?), by simpling inserting those B intervals one by one. In general, suppose that the last reconstruction of $\mathcal{T}$ happened when $N = N_0$. After $N_0/2$ updates, we simply rebuild $\mathcal{T}$ from scratch by incrementally inserting each interval currently in $\mathcal{I}$. Notice that now $\mathcal{I}$ can have at most $3N_0/2$ elements, so $\mathcal{T}$ can be reconstructed in $O(N_0 \log_B N_0)$ I/Os. This way, each of those $N_0/2$ updates bears merely $O(\log_B N_0)$ amortized I/Os. This ensures that the height of $\mathcal{T}$ remains $O(\log_B N)$ at all times.

Summarizing all the above discussion, we have:

Theorem 1. *Under the tall-cache assumption $M \geq B^2$, we can store N intervals in a structure that consumes $O(N/B)$ space, supports a stabbing query in $O(\log_B N + K/B)$ I/Os (where K is the number of reported intervals), and can be updated in $O(\log_B N)$ amortized I/Os per insertion and deletion.*

Remark. Arge and Vitter [1] showed that the tall-cache assumption can be removed such that all the complexities in Theorem 1 still hold even with $M = 2B$. They also removed the amortization such that each insertion and deletion can be handled in $O(\log_B N)$ I/Os in the worst case.

References

[1] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. of Comp.*, 32(6):1488–1508, 2003.

[2] M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, 1987.