Lecture Notes: Partially Persistent B-tree

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

Usually, a data structure is *ephemeral*, namely, once updated (with an insertion or a deletion), the structure has changed into a new version, such that its previous version (before the update) is lost permanently. In this lecture, we will discuss an interesting technique called *partial persistence* which allows us to preserve all the previous versions of a B-tree that has undergone a sequence of insertions and deletions.

Formally, we consider the following problem. Let S be a set of elements from the real domain $\mathbb{R}$. S is initially empty, and then modified by *updates*, each of which either inserts a new element into S or deletes an existing element from S. For convenience, we say that the i-th update happens at *timestamp* i. We denote by $S(i)$ the version of S at timestamp i (after performing the i-th update). Given a timestamp t and a value $q \in \mathbb{R}$, a *timestamp predecessor query* reports the predecessor of q in $S(t)$, namely, the largest element in $S(t)$ not exceeding q. The goal is to design a structure that consumes small space, can be updated efficiently along with S, and can answer timestamp predecessor queries efficiently. We will refer to the above as the *persistent predecessor problem*.

Naively, one can solve this problem by creating a B-tree on every $S(i)$ for all $i = 1, 2, ..., N$, where N is the total number of updates. Each query can be trivially answered in $O(\log_B N)$ I/Os. However, the space consumption and update overhead are terrible: all these B-trees can use $O(N^2/B)$ space, and creating a new B-tree for the next update may cost $O(N/B)$ I/Os. On the other hand, notice that two B-trees at consecutive timestamps share considerable common information. Thus, a natural question to ask is whether we can leverages such sharing to reduce the space and update cost. Next, we will see that this is indeed possible. In fact, we will lower the space all the way to the optimal bound of $O(N/B)$ (note that $\Omega(N/B)$ blocks are clearly needed to store N updates), and guarantee that each update be handled in $O(\log_B N)$ I/Os. Even better, we are still able to answer a query in $O(\log_B N)$ I/Os. Henceforth, we will assume that $B \geq 128$ is a multiple of 16.

1 Persistent B-Tree

Review: B-Tree. Let us start by slightly re-defining the B-tree for the purpose of this lecture. Let S' be a set of elements in $\mathbb{R}$, and T a B-tree on S'. All the leaves of T are at the same level. Each leaf node, if it is not the root, contains between $B/4$ and B elements in S'—referred to as *leaf elements*. Each element of S' is stored in one, and exactly one, leaf.

Consider now an internal node v of T with child nodes $u_1, u_2, ..., u_f$. If v is not the root, the value of f must satisfy $B/4 \leq f \leq B$; otherwise, it must hold that $f \geq 2$. For each u_i ($1 \leq i \leq f$), v stores a *routing element* e_i. Without loss of generality, suppose that $e_1, e_2, ..., e_f$ are in ascending order. For each $i \in [1, f]$, it must hold that *all* the leaf elements in $sub(u_i)$ are between e_i and e_{i+1} (defining dummy $e_{f+1} = \infty$). Note that routing elements are *not* required to be in S' (think: how would this affect predecessor search?). For each $i \in [1, f-1]$, we say that u_i and u_{i+1} are *neighboring siblings*.

Persistent B-Tree. Even though named a "tree", the *persistent B-tree*—designed by Becker et al. [1]—is actually an acyclic directed graph $\mathcal{T}$. A node of $\mathcal{T}$ is a *leaf* if no outgoing edge is adjacent

to it; otherwise, it is an *internal node*. A leaf node contains up to B *element-interval pairs*, each of which is in the form $(e, [t_{start}, t_{end}))$, where e is a real value and $[t_{start}, t_{end})$ is a time interval such that e belongs to $S(i)$ for each $i \in [t_{start}, t_{end})$. An internal node v of $\mathcal{T}$, on the other hand, can have at most B outgoing edges adjacent to it. Each outgoing edge is associated with an element-interval pair $(e, [t_{start}, t_{end}))$ stored in v. *Unlike* the leaf case, e is *not* required to be an element of $S(i)$ for each $i \in [t_{start}, t_{end})$. In fact, it is even possible that e does not belong to any $S(i)$ of $i \in [t_{start}, t_{end})$.

Given an element-interval pair $(e, [t_{start}, t_{end}))$, we refer to $[t_{start}, t_{end})$ as its *lifespan*. Also, the element-interval pair is said to be *alive* at each timestamp $i \in [t_{start}, t_{end})$. A node is *alive* at timestamp i if it contains at least one element-interval pair alive at i. Given any timestamp $i \in [1, N]$, $\mathcal{T}$ defines a tree $\mathcal{T}(i)$ as follows. First, $\mathcal{T}(i)$ includes all and only the nodes of $\mathcal{T}$ that are alive at i. Second, each node of $\mathcal{T}(i)$ contains all and only its element-interval pairs alive at i, and all and only the edges that those pairs correspond to.

We now finish the definition of the persistent B-tree with a crucial constraint: for each $i \in [1, N]$, $\mathcal{T}(i)$ must be a B-tree on $S(i)$.

2 Update

Next, we describe how to maintain the persistent B-tree $\mathcal{T}$ along with the updates on S. At the beginning, $\mathcal{T}$ is empty. Suppose that the update at timestamp 1 is to insert an element e into (an empty) S. We create a leaf node z with only one element-interval pair $(e, [1, \infty))$. In other words, $\mathcal{T}(1)$ is a B-tree with only a single node z. Inductively, assuming that $N-1$ updates have been processed, we will explain how to handle the N-th one.

Before the N-th update, $\mathcal{T}(N-1)$ is the "current" B-tree on $S(N-1) = S$. We follow the convention that, before the update starts, it always holds that $t_{end} = \infty$ for every element-interval pair $(e, [t_{start}, t_{end}))$ in $\mathcal{T}(N-1)$. Note that the convention implies $\mathcal{T}(N) = \mathcal{T}(N-1)$ at this moment. After the update, we will ensure the same convention, namely, $t_{end} = \infty$ for every element-interval pair $(e, [t_{start}, t_{end}))$ in $\mathcal{T}(N)$.

Re-balancing Operations. Given a node u in $\mathcal{T}$, we use $|u|$ to denote the number of element-interval pairs in u. Below we define three re-balancing operations that will be invoked during updates to help us maintain the structure's correctness:

- *Version Copy.* This operation is performed on a node u in $\mathcal{T}(N-1)$. Create a node u'. For each element-interval pair $(e, [t_{start}, \infty))$ in u, (i) change it to $(e, [t_{start}, N))$ if $t_{start} < N$, or discard it from u otherwise, and (ii) create an element-interval pair $(e, [N, \infty))$ in u'.

  Denote by $parent(u)$ the parent of u in $\mathcal{T}(N-1)$. If $parent(u)$ exists, then let $(e, [t_{start}, \infty))$ be the element-interval pair stored in $parent(u)$ for u. We change the pair to $(e, [t_{start}, N))$, add an outgoing edge from $parent(u)$ to u' (essentially, make u' a child of $parent(u)$), and associate the outgoing edge with a new element-interval pair $(e, [N, \infty))$. If, on the other hand, $parent(u)$ does not exist, it means that u was the root of $\mathcal{T}(N-1)$. We simply make u' the root of $\mathcal{T}(N)$.

  In any case, all the element-interval pairs in u' have their lifespans started at timestamp N. Such a node u' is said to be *fresh*.

- *Split.* This operation is performed *only* on a fresh node u such that $|u| \in [\frac{3}{4}B, \frac{11}{8}B]$. Sort the element-interval pairs of u in ascending order the elements they contain. Create a new node

u'. Move to u' the second half of the element-interval pairs in the sorted list of u. Note that both $|u|$ and $|u''|$ are now between $\frac{3}{8}B$ and $\frac{11}{16}B$.

If *parent*(u) exists, add an outgoing edge from *parent*(u) to u', and associate the edge with a new element-interval pair $(e', [N, \infty))$, where e' is the smallest element in the element-interval pairs of u'. If, on the other hand, *parent*(u) does not exist, then create a new node v with two outgoing edges to u and u', respectively. For the edge to u (or u', resp.), store in v an element-interval $(e, [N, \infty))$ (or $(e', [N, \infty))$, resp.) where e (or e', resp.) is the smallest element in the element-interval pairs of u (or u', resp.).

- *Merge.* This operation is performed *only* on two fresh nodes u, u'. It must hold that one of the two nodes has between $\frac{1}{4}B - 1$ and $\frac{3}{8}B - 1$ element-interval pairs, while the other one has between $\frac{1}{4}B$ and B element-interval pairs. We move all the element-interval pairs of u' into u, and accordingly, remove the edge from *parent*(u) to u', as well as the element-interval pair (stored in *parent*(v)) associated in the edge. If *parent*(u) is the root of $\mathcal{T}(N)$, and now has only one element-interval pair alive at timestamp N, make u the new root of $\mathcal{T}(N)$. If $|u| > \frac{3}{4}B$, split u.

Insertion. We now explain how to handle the N-th update if it inserts an element e into S. Recall that $\mathcal{T}(N-1)$ is a B-tree on the current S. We first find the leaf node z in $\mathcal{T}(N-1)$ that should accommodate e, and add an element-interval pair $(e, [N, \infty))$ into z. The insertion finishes if z currently has at most B element-interval pairs.

In general, a node *overflows* if it has more than B element-interval pairs. Given such a node u, we remedy its overflow as follows. First, perform a version copy u, which produces a new node u'. If $|u'| \in [\frac{3}{8}B, \frac{3}{4}B]$, then we are done. Otherwise, there are two possibilities:

- If $|u'| > \frac{3}{4}B$, split u'.
- If $|u'| < \frac{3}{8}B$, identify a neighboring sibling $\hat{u}$ of u' in $\mathcal{T}(N)$. Perform a version copy on $\hat{u}$ which produces a new node $\hat{u}'$. Merge u' and $\hat{u}'$.

The above steps may leave *parent*(u) overflowing, which is treated in the same manner.

Deletion. Suppose that the N-th update deletes an element e from S. First, find the leaf node z in $\mathcal{T}(N-1)$ that contains e. Change the element-interval pair $(e, [t_{start}, \infty))$ there to $(e, [t_{start}, N))$. The deletion finishes if z is the root of $\mathcal{T}(N)$, or it currently has at least $B/4$ element-interval pairs alive at timestamp N.

In general, a non-root node u of $\mathcal{T}(N)$ *underflows* if it has $B/4 - 1$ element-interval pairs alive at timestamp N. Given such a node u, we remedy its underflow as follows. First, perform a version copy u, which produces a new node u'. Identify a neighboring sibling $\hat{u}$ of u' in $\mathcal{T}(N)$. Perform a version copy on $\hat{u}$ which produces a new node $\hat{u}'$. Merge u' and $\hat{u}'$.

The above steps may leave *parent*(u) in one of the following 3 states:

- Neither overflowing nor underflowing. The deletion finishes.
- Overflowing but not underflowing. Handle the overflow as described earlier for insertion.
- Underflowing (perhaps also overflowing). Handle the underflow as described above, and ignore the overflow.

Update Cost. Each re-balancing operation requires $O(1)$ I/Os. Since $T(N)$ is a B-tree with height $O(\log_B N)$, each insertion/deletion can be performed in $O(\log_B N)$ I/Os.

3 Query

By definition, the persistent B-tree contains a B-tree $\mathcal{T}(i)$ for every timestamp $i \in [1, N]$. Hence, to answer a predecessor query at timestamp t, we simply find the root of $\mathcal{T}(t)$, and then, answer the query in $\mathcal{T}(t)$. The overall cost is $O(\log_B N)$. Note that we have not discussed how to find the root of $\mathcal{T}(t)$. We leave to you to figure out how to do so efficiently.

4 Space

It remains to prove that the persistent B-tree uses $O(N/B)$ space. First, the following property is easy to verify:

Lemma 1. *When a node is created from an overflow or underflow, it is a fresh node with between $\frac{3}{8}B$ and $\frac{3}{4}B$ elements.*

We say that an *insertion* happens in a node u if an element-interval pair is inserted in u, and that a *logical deletion* happens in u if an element-interval pair $(e, [t_{start}, t_{end}))$ has its ending time t_{end} changed from ∞ to a finite value. Also, we say that leaves are at *level* 0, and that, in general, the parent of a level i node is at level $i + 1$.

There are in total N insertions and logical deletions into the leaf nodes of $\mathcal{T}$. Lemma 1 implies that at least $B/8$ insertions and/or logical deletions must be made into a fresh leaf node before it overflows or underflows. Therefore, there are at most $N/(B/8)$ overflows and underflows at the leaf level. This means that there are $2N/(B/8) = N/(B/16)$ leaf nodes because each overflow/underflow creates at most 2 leaves.

In treating the overflow/underflow of a leaf node, at most 4 insertions and/or logical deletions are made into nodes of level 1. It follows that the total number of insertions and logical deletions at this level is at most $4N/(B/16) = N/(B/64)$. Our earlier argument based on Lemma 1 shows that there are at most $\frac{N}{(B/16)(B/64)}$ nodes at level 1. In general, at most $\frac{N}{(B/16)(B/64)^i}$ nodes are created at level i. When $B \geq 128$, the overall space is linear.

References

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