CSC6210: Advanced multidimensional search — Fall 2009
CSE department, Chinese University of Hong Kong

Lecture 2: Persistent B-tree

By Yufei Tao (taoyf@cse.cuhk.edu.hk) Last revision: Sep 6, 2009

Forewords

This lecture considers the following *orthogonal segment intersection* problem. Let S be a set of horizontal segments in the 2d space. We need to pre-process S so that, given any vertical segment q, all the segments of S intersecting q can be reported efficiently. In Figure 1, for example, segments s_1 and s_2 are to be reported.

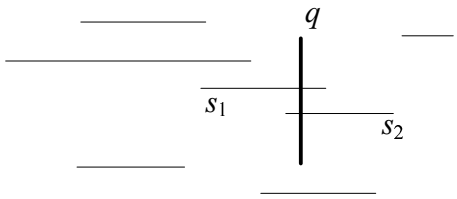

Figure 1: Orthogonal segment intersection

We will learn a structure called the *persistent B-tree*, which optimally occupies $O(n/B)$ space and answers any query in $O(\log_B n + k/B)$ I/Os, where $n = |S|$ and k is the result size.

1 A naive solution

Denote by x_q the x-value of q, and by r_q its projection on the y-axis. For each segment $s \in S$, let $I_s = [x_s^{start}, x_s^{end}]$ be its projection on the x-axis, and y_s its y-value. We say that s is *alive* at any $x \in I_s$, and I_s is the *lifespan* of s.

Denote by $S(x)$ the set of segments in S that are alive at x. So the objective is to report the segments of $S(x_q)$ whose y-values fall in r_q. This would be easy if we *had* a B-tree that indexes the y-values of all the segments in $S(x_q)$. To support all queries, we need a B-tree, denoted as $\mathcal{T}(x)$, indexing the $S(x)$ at every possible x. Refer to x as the *version* of $\mathcal{T}(x)$. The number of trees may seem to be infinite at first, but we can bound it by observing that a new tree is required at x only if some segment in S starts or ends at x. Hence, $O(n)$ trees are enough, each responsible for a set of versions that form a continuous interval, which we call the tree's *version interval*.

We answer a query as follows. First, identify $\mathcal{T}(x_q)$. This can be done in $O(\log_B n)$ I/Os by indexing the version intervals of all B-trees with another B-tree. We then query $\mathcal{T}(x_q)$ with r_q, and report all the qualifying segments in $O(\log_B n + k/B)$ I/Os. The query cost is very good but the problem is space. Since each tree occupies $O(n/B)$ pages, the total space is $O(n^2/B)$, which is prohibitively expensive.

We will fix the problem with the (partial) *persistence technique*, also known as the *multiversion technique* [1, 2, 3]. The key is not to store the trees separately as before. Intuitively, two consecutive B-trees may be very similar, so too much space was wasted by duplicating their common parts. The persistence technique stores only the "delta" between two consecutive trees, thus reducing the space

dramatically.

2 The persistent B-tree

The structure. As mentioned earlier, our goal is to, conceptually, have a B-tree $\mathcal{T}(x)$ at every x-value x (which is the version of $\mathcal{T}(x)$). A persistent B-tree encodes all these B-trees in a space economical manner. It allows entries of different versions to be stored in the same page. For this purpose, a conventional B-tree entry e needs to be augmented with a *lifespan* in the form of $[x_e^{start}, x_e^{end})$. This is to indicate that e belongs to every B-tree $\mathcal{T}(x)$ with $x \in [x_e^{start}, x_e^{end})$. Equivalently, $\mathcal{T}(x)$ consists of all and only the entries whose lifespans contain x, namely, *alive* at x.

Each page can accommodate at most B entries. An important property of the persistent B-tree is that every page contains either no or at least $B/4$ entries that are alive at any x-value. Hence, for any x, $\mathcal{T}(x)$ has a page fanout $\Omega(B)$. This allows us to search $\mathcal{T}(x)$ in the same time bound as querying a normal B-tree.

Another property is that, in building the persistent B-tree, every new page must contain between $\frac{3}{8}B$ and $\frac{7}{8}B$ entries. This ensures that at least $\Theta(B)$ insertions and/or deletions are needed before the page spawns another one, which is crucial for bounding the space.

The construction algorithm. A persistent B-tree can be built with a *planesweep* approach. Imagine a vertical sweeping line ℓ that is initially placed to the left of all the segments in S. Then, ℓ is moved towards right, until all data segments are on its left. In the meantime, we keep building $\mathcal{T}(x)$ at the current position x of ℓ. So at the end of the sweep, we have a B-tree for every x-value.

We will discuss the details under a *general situation* of S. Namely, at every x-value, either at most one data segment starts, or at most one segment ends, but not both. Otherwise, we can apply an infinitesimal perturbation to make it so. This is a common trick to avoid boundary cases that are not difficult to handle, but complex enough to cause distraction from discussing the core technique.

There are only two operations:

- *Insertion.* When ℓ hits a data segment s, insert y_s in $\mathcal{T}(x_s^{start})$.
- *Deletion.* When ℓ reaches the end of a data segment s, delete y_s from $\mathcal{T}(x_s^{end})$.

We first explain how to do an insertion. To simply notations, let $x = x_s^{start}$, and denote by $|u|$ the number of entries in a node u. The insertion starts by identifying the leaf node u in $\mathcal{T}(x)$ that should accommodate y_s. If the page of u is not full (i.e., having less than B entries), y_s is added to u with a lifespan $[x, \infty)$, which completes the whole insertion.

Otherwise, u *overflows*. We then perform a *version split* which involves two steps: (i) copy the entries of u alive at x to a new page u', where all lifespans are set to $[x, \infty)$; (ii) *kill* the alive entries in u at x, by terminating their lifespans at x. If $|u'| > \frac{7}{8}B$, split it into itself and u'' by distributing the entries evenly. On the other hand, if $|u'| < \frac{3}{8}B$ (a *strong underflow*), version split a sibling of u in $\mathcal{T}(x)$ into u''. Merge u'' into u' if $|u'| + |u''| \leq \frac{7}{8}B$; otherwise, distribute the entries evenly between u' and u''. It is not hard to verify that $|u'|$ and $|u''|$ both fall between $\frac{3}{8}B$ and $\frac{7}{8}B$ after the above steps.

We also need to update $parent(u)$ in $\mathcal{T}(x)$ accordingly. The idea is that the reference to a child that has been version split needs to be killed. On the other hand, for every new-born child, a reference is added with a lifespan $[x, \infty)$. This may cause $parent(u)$ to overflow too, which is handled similarly, and propagates the changes to the upper levels.

Deletion is performed similarly. Let $x = x_s^{end}$. First, kill the entry of s in the relevant leaf node u of $\mathcal{T}(x)$. If the page of u still has at least $B/4$ entries alive at x, the deletion finishes. Otherwise, u *underflows*, which is treated in the same way as a strong underflow. Finally, the changes are propagated to $parent(u)$ and further upwards if necessary.

Space complexity. Number the node levels bottom-up, setting the leaf level at level 0. A new leaf node is created only when an existing one overflows or underflows. Since a leaf node must take $\Omega(B)$ insertions and/or deletions to generate an overflow or underflow, the total number of leaf nodes is $O(n/B)$. By the same argument, the total number of level-i, $i \geq 1$, nodes is $O(n/B^i)$. Therefore, the overall space is $O(n/B)$.

We thus arrive at:

Theorem 1. *Given a set S of n horizontal segments in a 2d space, we can pre-process it into a persistent B-tree that occupies $O(n/B)$ pages. Given any vertical segment q, all the segments of S intersecting q can be reported in $O(\log_B n + k/B)$ I/Os, where k is the result size.*

Remark 1. The persistent B-tree we described can be applied directly in the internal memory, by setting B to a constant at least 16. The resulting structure requires $O(n)$ space and $O(\log n + k)$ query time.

Remark 2. The construction algorithm we described requires $O(n \log_B n)$ I/Os in total. This is not attractive at all. Using another technique called *distribution sweep*, we can lower the cost to $O(\frac{n}{B} \log_B n)$.

References

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

[2] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

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