COMS E6998: Advanced Data Structures (Spring'19) Mar 28, 2019

Lecture #9: Cell Probe LBs for Dynamic Range Counting

Instructor: *Omri Weinstein* Scribes: *Weston Jackson*

1 Last Time

Orthogonal Range Counting (ORC): return all points (or sum of weights of all points) in a rectangle R:

$$\sum_{(x,y)\in R} w_{xy}$$

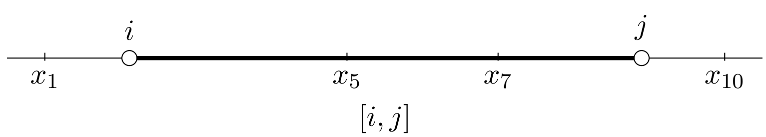
Previously, we used layered range trees to solve in time:

- Static: space $s = O(n \lg^{d-1} n)$, $t = O(\lg^{d-1} n)$ $\quad \forall\, d \geq 2$
- Dynamic: $t_u = O(\lg^{d} n)$, $t_q = O(\lg^{d-1} n)$ $\quad \forall\, d \geq 2$

Fort he static case, we used fractional cascading in the last layer to save one lg factor. We pay $\lg n$ overhead for dynamization, which adds this factor back for *both* update and query times (via exponential-blocking/segment trees, as ORC is *decomposable*). It is possible to do better by exploiting the *tree structure* of (static) layered range trees, losing the logarithmic factor only in *update time* – this yields a fully-dynamic ORC DS with $\hat{t_u} = O(\lg^d n), t_q = O(\lg^{d-1} n)$ via weight-balanced (a.k.a $BB[\alpha]$) trees[1].

Question : Can we have a similar (asymmetric) speedup in either update or query time for $d = 1$?

1D-ORC/ Partial Sums (PS_n): Updates are insertions into the number line. Queries ask us to report all points in some interval $[i, j]$:



1D ORC is equivalent to the Partial-Sums problem PS_n: Our updates just set the index $A[i] \leftarrow \{0, 1\}$ (or some larger weights in general). For queries, we define $PREFIX()$ function as the following:

$$PREFIX(i) = \sum_{j \leq i} A[j]$$

For $QUERY(i, j)$, we make two $PREFIX()$ calls, and just return:

$$PREFIX(j) - PREFIX(i)$$

[1]See e.g., Sec 3.4 here: https://pdfs.semanticscholar.org/841a/31780b7e8f4de224fac06181321ca2ea807e.pdf

Static Case. The static PS problem is almost trivial: we can just precompute the answers! Directly store $B[i] := \sum_{j \leq i} A[j]$. On query, $B[i] = PREFIX(i)$ and thus we can answer in constant time. For example, if A is the following:

1	0	1	0	0	1	0	0	1	1

Then B would be:

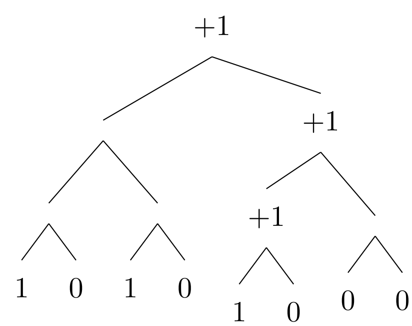
1	1	2	2	2	3	3	3	4	5

The space for precomputing answers is in general $O(u)$ words, where $u =$ universe size, so if $u \gg n$, this is no longer linear space. However, we observe that we can make the space linear ($s = O(n)$) at the price of $t = O(\lg \lg u)$ search time, by precomputing the answers for each of the n keys – We can then use *Predecessor* search to get the partial sum up to that key.

Alas, this data structure has heavy preprocessing and cannot be cheaply maintain dynamically, so the following question remains: how do we maintain this structure when inserting new points?

1.1 Dynamic Partial Sums

The simple idea, essentially equivalent to (vanilla) 1D Range Trees, is to keep a tree where the leaves point at the array A. Each node in the tree keeps track of the partial sums in its left and right subtrees. Thus, when we insert a new point x, we just update each node n along the path from root to leaf:



Clearly, both update and query times of this dynamic data structure are $t_u = t_q = O(\lg n)$. Perhaps surprisingly, we can do updates significantly faster, while maintaining the same (logarithmic) query time (which we shall soon see is *optimal*):

Theorem 1. *There exists a dynamic partial sum data structure with:*

$$t_q = O(\lg n), \hat{t_u} = O(\sqrt{\lg n})$$

Main ideas: (1) Delay updates by buffering. (2) Exploit the self-reducibility of PS_n

Claim 2. *Suppose there exists a data structure $\mathcal{D}_L$ for PS_{2^L} (a smaller array). Suppose it has update and query time t_u^L, t_q^L. Then we can design a data structure for PS_n using:*

$$t_u = O(t_u^L \cdot \frac{\lg n}{L}), t_q = O(t_q^L \cdot \frac{\lg n}{L})$$

Proof. We maintain a tree as before, but each node has fan out of size 2^L. Then the total height of the tree is $\frac{\lg n}{L}$. Each node also maintains the smaller data structure $\mathcal{D}_L$.

- $INSERT(x)$: At each node, insert x into small partial sum data structures $\mathcal{D}_L$.
- $QUERY(x)$: Traverse tree, query each node for partial sums in children in time $\frac{\lg n}{L}$.

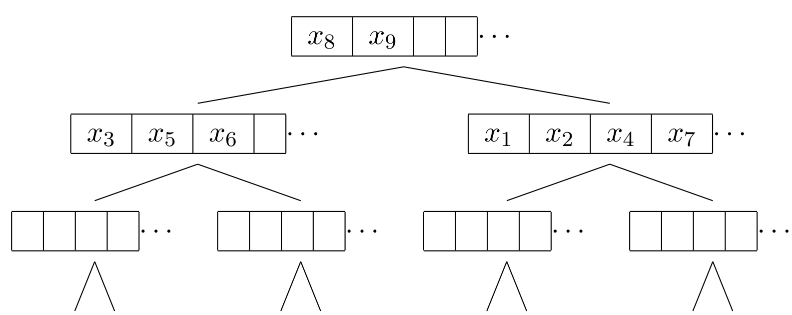
The time to query and update each node is just t_q^L and t_u^L, thus the total query and update time is:

$$t_u = O(t_u^L \cdot \frac{\lg n}{L}), t_q = O(t_q^L \cdot \frac{\lg n}{L})$$

□

1.2 Buffer Trees

Note that there is room for improvement in our partial sum tree data structure. Our word size is $\lg n$ and key size is L, but we don't exploit the fact that we can store lots of keys in a single word. (intuitively, the range tree solution is suboptimal in the sense that when it performs an update, it inserts w bits (a key) into $\sim w$ levels of the tree, i.e., only a "single bit on average" to each layer, whereas each node of the tree can store w bits of information, hence intuitively the "bandwidth" is w^2 bits). Indeed, we can improve on this using *Buffer Trees*:



- Build a binary tree on top of the 2^L-sized array. Keep a buffer at each node of size $w = \lg n$. $\forall$ node $\in \mathcal{D}_L$ maintains a buffer of most recent n $\Theta(\frac{w}{L})$ updates (indeed, note that a key in this subtree requires only L bits to describe, which is the key point we are leveraging here). If the root buffer is not full, just insert into the buffer.
- $INSERT(x)$:
  (1) If current node buffer is not full, insert x into the buffer (do not reflect this new key in the partial sums of the left and right subtrees).
  (2) If current node buffer is full, flush the buffer and distribute the updates to the children. Recompute partial sums of both children ($O(1)$). Recurse if necessary.
- $QUERY(x)$:
  Traverse the tree and collect the partial sums + buffers at each node. Traversing the tree takes time $\lg(depth\ D_L) = L$, thus total search time in the *original* tree is:

$$t_q = O(t_q^L \cdot \frac{\lg n}{L}) = O(L \cdot \frac{\lg n}{L}) = O(\lg n)$$

Amortized insertion analysis:

$$Cost(t_u^L) = O(1) + \text{Amortized Cost (flushing)}$$

The cost of flushing a buffer is $O(1)$ and the buffer flushes only with $O(\frac{L}{w})$ frequency. A single inserted key cannot trigger more than L flushes total when going down the tree. Thus the amortized cost is: $O(\frac{L^2}{w})$:

$$Cost(t_u^L) = O(1) + O(\frac{L^2}{w})$$

To calculate the overall amortized cost in the original tree:

$$t_u = \frac{\lg n}{L}(O(1) + O(\frac{L^2}{w})) = \frac{\lg n}{L} + \frac{L \lg n}{w}$$

Choosing $L = \sqrt{w}$ and using $w = O(\lg n)$:

$$t_u = O(\sqrt{\lg n})$$

OPEN: Is this optimal? It is conjectured that $t_u = o(\sqrt{\lg n}) \implies t_q = \omega(\lg n)$.

2 Lower Bounds: Chronogram Method

Theorem 3 (FS '89)**.** *For all dynamic data structures for PS_n, $t_q \geq \Omega(\lg_{wt_u} n)$. This implies that:*

$$max\{t_u, t_q\} \geq \Omega(\frac{\lg n}{\lg \lg n})$$

Idea 1: Do a series of random insertions $\in_R \{0, 1\}$ into random locations of array A, and then perform a random query. The high-level approach is to show that after n random updates to A, a random PS_n query $q \in_R [n]$ must read a lot ($\sim \lg n$) memory cells.

To this end, divide the n random updates into geometrically decaying epochs $U_k \cdots U_1$:

$$U_k = \boxed{u_1 \mid u_2 \mid u_3 \mid \cdots \mid \;\mid\;} \cdots \; U_i = \boxed{\;\mid\;\mid\;\mid\;} \cdots \; U_1 = \boxed{\;}$$

In each epoch $|U_i| = \beta^i$, where $\beta = (t_u \cdot w)^3$ and $k = \Theta(\lg_\beta n)$. We then insert β^i random updates into evenly spaced locations:

$$\forall j = 1 ... \beta^i, A[j \cdot \frac{n}{\beta^i}] := u_j$$

where $u_j \in_R \{0, 1\}$. For example:

- U_1 updates $A[0\frac{n}{\beta}], A[1\frac{n}{\beta}], A[2\frac{n}{\beta}] \cdots$
- U_2 updates $A[0\frac{n}{\beta^2}], A[1\frac{n}{\beta^2}], A[2\frac{n}{\beta^2}] \cdots$
- U_k updates $A[0\frac{n}{\beta^k}], A[1\frac{n}{\beta^k}], A[2\frac{n}{\beta^k}] \cdots$

etc. Remember that the updates are processed from $U_k \to U_1$.

Claim 4. *Geometric decay reduces a dynamic problem on n updates to roughly $\lg n$ independent state problems.*

Claim 5. *The only memory cells in the data structure that reveal substantial information about U_i are cells written during that epoch.*

Let $D(U_i)$ be the memory state of the data structure after epoch U_i. Let $A_i :=$ the set of memory cells last written during U_i. This is equivalent to a partition of the memory state into $\lg n$ colors. For example if we denote A_i as red (r), A_{i-1} as blue (b), A_{i-2} as green (g):

$D(U_{i-2}) =$

r	b	r	b	r	r	r	b	g	r

The idea is that a certain number of registers A_i for epoch U_i must be queried to reflect the events from epoch U_i. To show this, consider how many bits of information about U_i can be revealed by the past and future epochs:

- Past $A_{>i}$: These reveal no information about U_i because past updates are independent in that they happened beforehand.
- Future $A_{<i}$: These are not necessarily independent from U_i. Its possible a memory cell in the future copied some memory cell that was written during the epoch U_i. But considering that the number of updates decays geometrically, very few cells should be written in the future.

Calculating the number of cells that can be written after U_i

$$\begin{aligned}
\sum_{j=1}^{i-1} |U_j| \cdot t_u \cdot w &= \sum_{j=1}^{i-1} \beta^j (t_u w) \\
&\leq 5\beta^{i-1} \cdot t_u w && \text{Since } \beta_j \text{ is decating} \\
&<< \beta_i = |U_i| && \text{Since } \beta_i = (t_u \cdot w)^3
\end{aligned}$$

Lemma 6. *For large epochs (any epoch with size > a small constant), we have that:*

$$\mathbb{E}_{q,U}[|D(q) \cap A_i|] \geq \Omega(1)$$

when $D(q)$ reads t_q memory cells on query q

Note that this implies the total size of the data structure over random query is at least:

$$\begin{aligned}
\mathbb{E}_{q,U}[|D(q)|] &\geq \sum_{i=1}^{k} \mathbb{E}[|D(q) \cap A_i|] && A_i \text{ are disjoint} \\
&= \sum_{i=1}^{k} \Omega(1) \\
&= \Omega(\lg_\beta n)
\end{aligned}$$

Proof. Assume for purpose of contradiction that the Lemma is false, and there is an epoch where $\mathbb{E}_{q,U}[|D(q) \cap A_i|] = o(1)$. Let epoch U_i be this epoch. Then 99% of partial sum queries $q \in [n]$ do not read cells A_i. Consider all other epochs fixed. Alice's input is all epochs $U_k \cdots U_1$, while Bob's input is all epochs except for U_i. We construct an impossible compression scheme such that we can encode β^i random updates in $< \beta^i$ bits.

	Alice	Bob
Input	$U_k, U_{k-1}, \cdots, U_i, \cdots U_1$	$U_k, U_{k-1}, \cdots, (?), \cdots U_1$

Idea 1: Alice sends Bob all updated contents $A_{<i} = o(\beta_i)$.
Idea 2: Alice sends parity $\in \{0, 1\}$ for 1% of queries that touch A_i.

Decoding: Bob simulates his data structure for epochs $U_{i+1} \cdots U_k$ to get cells $A_{>i}$. Bob then updates his data structure with the contents of $A_{<i}$ from Alice. The only cells Bob doesn't know are A_i. Bob uses parity of queries from Alice (for 1% of queries that touch A_i) and existing data structure to reconstruct the partial sums for any new query.

Complexity: The size of Alice's first message is the number of cells written by epochs $U_1 \cdots U_{i-1}$ which can be encoded in $\leq \frac{\beta^i}{4}$ bits. The size of Alice's parity messages are:

$$\lg \binom{\beta_i}{\beta_i/100} \approx \beta_i \frac{\lg 100}{100} < \frac{\beta_i}{4}$$

Thus, the total size of Alice's messages is $\frac{\beta_i}{4} + \frac{\beta_i}{4} < \beta_i$. This is a contradiction, as the encoding should be at least β_i.

$\square$