Lecture Notes: First-in-first-out Indexing

Yufei Tao
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

22 Apr, 2012

This lecture discusses a stream model called the *first in first out* (FIFO) model, which generalizes the sliding window model we are familiar with. In the FIFO model, we are looking at an unbounded stream of elements, and at any moment of time, are interested in only a set S of most recently received elements. This sounds similar to the sequence-based sliding window model, where S has a fixed size, because the arrival of a new element always implies the deletion of the oldest element in S. FIFO model, on the other hand, allows an arbitrary interleaving of insertions and deletions (namely, there can be any number, including 0, of insertions between two consecutive deletions). The only constraint is that elements must be deleted in the same order as they arrive, and hence the name FIFO. The size of S, which we denote as n, can therefore vary arbitrarily.

The sequence-based sliding window model is apparently a special case of the FIFO model. The same is true for the *time-based sliding window* model, in which elements arrive at the end of every second[1], and expire (and hence, deleted) after t seconds, where t is a parameter. S includes all the elements that have not expired. Since there is no control on how many elements can arrive in a second, the size of S can fluctuate significantly. For elements that arrive at the same second, there is no relative ordering among them, but we can apply an arbitrary ordering (say, but their ids) to fit the time-based sliding window into the FIFO model.

In some applications, we want to support queries on the elements in S. For example, imagine that S includes the current locations of taxi cabs (each of which informs a server about its current location, say, every minute). A useful query in practice asks "where is my nearest taxi cab?" If we assume Euclidean distance as the distance metric, then this query is essentially nearest neighbor search, but performed on the (2d) points in S. An interesting problem is to maintain a structure on S to answer such queries fast. Of course, our structure must support *both* insertions and deletions efficiently, too.

Generalizing the above scenario, most problems can be studied in the *FIFO indexing* scenario. In such a scenario, we want to maintain a structure on a dynamic set S of elements. An insertion can add any new element to S, but a deletion is only allowed to remove the element that was inserted the longest time ago. The structure should enable us to answer a query efficiently. Note that we have not specified the details of the query. In fact, the technique we are about to discuss is general enough to work for any decomposable problem (e.g., nearest neighbor search). The only requirement is that that we know how to build a *static* structure for solving the problem – very much like the logarithmic method!

1 Converting a static structure to a FIFO index

Assuming that we have designed a static structure for solving the underlying problem, next we describe a technique for obtaining a FIFO index on the same problem. The technique was proposed

[1]Here, we assume that a second is the time unit, but the idea obviously carries over to all time units.

by Sheng and Tao [1].

Structure. Given two elements $e_1, e_2 \in S$, we say that e_1 is *newer* than e_2 if e_1 arrived later than e_2; otherwise, e_1 is *older*. At all times, we divide S into $h^- + h^+ + 2$ disjoint subsets, where h^- and h^+ are integers maintained by our algorithm, and satisfy:

$$\begin{aligned} h^- &= O(\lg n) \\ h^+ &= O(\lg n) \\ h^- &\geq h^+. \end{aligned} \tag{1}$$

Recall that n is the size of S. We denote those subsets as $S_0^-, ..., S_{h^-}^-, S_{h^+}^+, ..., S_0^+$, arranged in *anti-chronological order*, namely, elements in S_0^- are the oldest and those in S_0^+ the newest. For each $i \in [0, h^+]$, it holds that $|S_i^+|$ is either 0 or 2^i (note: a subset can be empty). Similarly, for each $i \in [0, h^-]$, $|S_i^-|$ is either 0 or 2^i. The only exception is $S_{h^-}^-$, which is always non-empty as long as $n > 0$. Each non-empty subset is indexed by a static structure. As a special case, if $n = 0$, then no subset exists, and $h^- = h^+ = -1$.

Finally, we also maintain a queue of all elements in S, sorted by the order they arrive. Call this queue the *chronological queue*.

Query. This is straightforward: simply search all structures and combine the results.

Insertion. To insert an element e, we adopt the same approach as logarithmic rebuilding. Find the smallest $i \in [0, h^+]$ such that S_i^+ is empty. If no such i exists (i.e., $S_{h^+}^+, ..., S_0^+$ are all non-empty), increase h^+ by 1, and set i to the new h^+. We *merge* all the elements of $S_{i-1}^+, ..., S_0^+$ together with e to S_i^+ (if $i = 0$ then simply create $S_i^+ = \{e\}$), and build a (static) structure on S_i^+. Accordingly, $S_{i-1}^+, ..., S_0^+$ are emptied.

If h^+ has been increased, it may now be greater than h^- (by 1), which violates (1). In this case, we set h^- to h^+, rename $S_{h^+}^+$ to $S_{h^-}^-$, and reset h^+ to -1.

Deletion. Deleting the oldest element e is carried out in a reverse manner. First, find the smallest $i \in [0, h^-]$ such that S_i^- is non-empty. Note that e is definitely in S_i^-. After discarding e, S_i^- has $2^i - 1$ elements left. If $i > 1$, *split* these elements into $S_0^-, ..., S_{i-1}^-$, obeying their chronological order requirement mentioned earlier (this can be done in $O(2^i)$ time using the chronological queue). Construct a structure on each of $S_0^-, ..., S_{i-1}^-$, and empty S_i^-.

If i equals h^-, we need to decrease h^- by 1. Now, h^- may become less than h^+ (by 1). In this case, set h^- to h^+, rename $S_{h^+}^+$ to $S_{h^-}^-$, and decrease h^+ to the largest j such that S_j^+ is non-empty. If no such j exists, set h^+ to -1.

2 Analysis

We now analyze the performance guarantees of the FIFO index obtained by the technique described in the previous section. We will focus on the update cost, leaving out the query cost and space consumption (whose analysis is trivial for most problems). The update cost depends on the construction time of the static structure deployed. We characterize the construction time as at most $t \cdot U(t)$, if the underlying dataset has t elements. For example, a B-tree on t real numbers can be built in $O(t \lg t)$ time, in which case $U(t) = O(\lg t)$. We will prove:

Theorem 1. *In the FIFO index described in Section 1, the insertion and (if applicable, also) deletion of an element e require $U(2n_e) \cdot O(\lg n_e)$ amortized time, where n_e is the size of S at the time e is inserted.*

Note that the $U(n_e) \cdot O(\lg n_e)$ is the *total* amortized cost of inserting and deleting an element e. How the cost is divided into insertion and deletion time is irrelevant: e.g., one can allocate all the $U(n_e) \cdot O(\lg n_e)$ to insertion, and claim that a deletion incurs zero cost.

Recall that our algorithm divides S into subsets $S_0^-, ..., S_{h^-}^-, S_{h^+}^+, ..., S_0^+$. We say that the elements in $S_{h^+}^+ \cup ... \cup S_0^+$ are in the *insertion phase*, while those in $S_0^- \cup ... \cup S_{h^-}^-$ in the *deletion phase*. Let us start with a key observation:

Lemma 1. *At any time, if x elements are in the insertion phase, then less than $2x$ elements are in the deletion phase.*

Proof. The lemma is vacuously true if $n = 0$. For $n > 0$, $S_{h^-}^-$ is non-empty; hence, $x \geq 2^{h^-}$. On the other hand, the insertion phase can have at most $2^0 + 2^1 + ... + 2^{h^+} < 2^{h^+ + 1} \leq 2^{h^- + 1} \leq 2x$. □

Now we give another crucial fact about e:

Lemma 2. *At any time, e is in a subset of size less $2n_e$.*

Proof. If e is still in the insertion phase, all the elements in the deletion phase must be older than e. Hence, the deletion phase has no more than n_e elements. By Lemma 1, the entire insertion phase has less than $2n_e$ elements, implying that the subset where e is has less than $2n_e$ elements.

Consider the moment when the subset of e migrates to the deletion phase (i.e., due to renaming). By the previous paragraph, the subset has size less than $2n_e$ before the migration. Since migration does not affect the subset size, e is still in a subset of size less than $2n_e$. From now on, e can only move to smaller subsets, thus completing the proof. □

Now we are ready to prove Theorem 1. The cost of maintaining our FIFO index comes from merging and splitting subsets (in the insertion and deletion phases, respectively). If a merged subset has 2^i elements for some i, the merging costs

$$2^i \cdot U(2^i)$$

time. We can amortize the cost over all the 2^i elements that are now in the merged subset, so that each element bears $U(2^i)$ time at most. By Lemma 2, e bears at most $U(2n_e)$ time for each of the merges involving e.

Similarly, splitting a subset of size $2^i - 1$ (for some i) takes

$$\sum_{j=0}^{i-1} 2^j \cdot U(2^j) \leq (2^i - 1) \cdot U(2^i)$$

time. Amortizing the cost to the $2^i - 1$ elements in the divided subset, we charge at most $U(2^i)$ cost on each of them. By Lemma 2, e bears at most $U(2n_e)$ time for each of the splits involving e.

How many times does e need to bear the cost of $U(2n_e)$? During the insertion (deletion) phase, e moves into subsets of increasing (decreasing) sizes. By Lemma 2, we know that e appears in at most $O(\lg n_e)$ subsets in each phase. This completes the proof of Theorem 1.

References

[1] C. Sheng and Y. Tao. FIFO indexes for decomposable problems. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 25–35, 2011.