Differentially Private Quantile Trees

Quantile Trees are a general purpose mechanism to compute differentially private (DP) quantiles, offering benefits such as distributed computation, unlimited quantile queries, easily verifiable privacy, etc. For certain use cases, they may be outperformed by other mechanisms, but their generality makes them ideal for the DP Building Blocks library.

Content

How it Works

At its core, a Quantile Tree is a simple tree data structure of height h whose leaf nodes split the dataset's domain (that is to say the range of numbers between some provided lower bound l and upper bound u) into b^h **buckets**, where b is the number of children per non-leaf node. Each node of the tree is identified by a unique index (illustrated by the red fields in Figure 1) and carries an internal **counter** (the blue fields in Figure 1). Think of a leaf node's counter as the number of elements in the bucket associated with the node. The counters at higher levels in the tree keep track of the total sum of elements in the leaf nodes of their respective subtree.

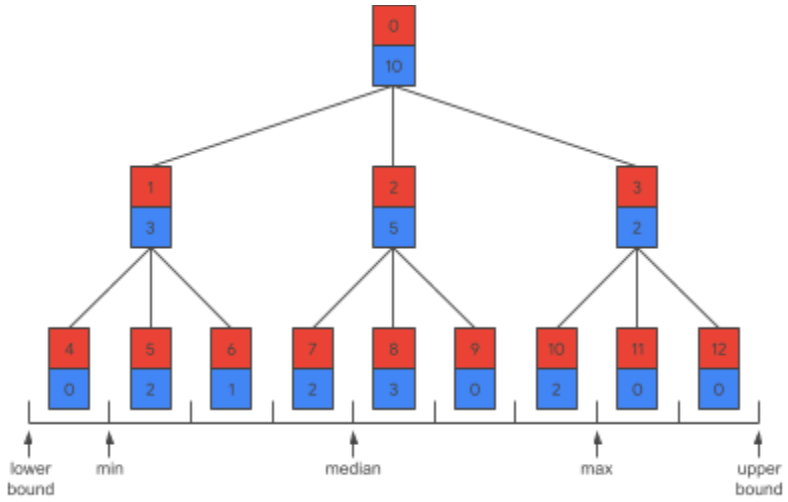

Figure 1: Example of a Quantile Tree for $h = 2$ and $b = 3$. The index of each node is stored in the red section of the nodes while the counters are stored in the blue section.

Given this simple data structure we can:

- **Compute a quantile** by looking for the bucket containing it. This bucket can be found efficiently by a simple tree search from the root to the respective leaf. All information needed to perform this search is contained in the counters. See this section for more information.
- **Add elements** by traversing the tree from the root to the respective leaf and incrementing each counter along the path by 1.
- **Merge Quantile Trees** by adding all counters of the same index.

Moreover, we can make a Quantile Tree DP by adding an appropriate amount of noise, either via the Laplace or the Gaussian mechanism, to the counter of each of its nodes. The exact amount of noise can be found in the privacy proof.

How to Make it Faster

Initializing and noising a complete Quantile Tree is expensive. So, we only perform these tasks when necessary. The idea is simple. Instead of storing the nodes of a Quantile Tree in a single data structure of fixed size , e.g., an array, we store them in two separate data structures, one for raw counters and one for noisy counters, and we populate these data structures **lazily** using **hashmaps**, mapping node indices to counter values.

Quantile Trees can be in one of two states. The first state is the **aggregation state**. While in this state, elements are added to the data structure by inserting them into one of the hashmaps, let's call this one hashmap A. The insertion process is identical to that described in the previous section, but only keeps track of counters greater than 0.

Once the first quantile query is made, the Quantile Tree switches to the **query state**. This means that no more elements can be added, effectively freezing hashmap A. From now on, any query we answer we answer by looking at the noised values in hashmap B. If B already contains the noised count of a node we are looking up, we use that count. Otherwise we look up the raw count in A and copy that value with an appropriate amount of noise to B, where it is stored for future reference.

Noisy Quantile Search

For the sake of simplicity, we try to keep the searching for a particular quantile as easy as possible. Although the search is mostly straight forward, we need to pay attention to potential inconsistencies due to the added noise, e.g., fractional counters, negative counters and counters not matching the sum of their children's counters. The following is a pseudocode implementation of the search algorithm we use:

1. **Let** x be the root node and $q \in [0, 1]$ the quantile we are looking for
2. **While** x is not a leaf node
3. **Let** Y be the set of children y of x whose noisy counter c_y is positive

4. **Let** $t := \Sigma_{y \in Y} max\{0, c_y\}$ be the sum of counters in Y (ignoring negative values)
5. **Let** $Y' \subseteq Y$ be the set of children y whose noisy counter c_y is greater than $\alpha \cdot t$ (the idea of α is to filter out nodes that are most likely empty)
6. **If** Y' is empty
7. **Set** $q := 0.5$ (i.e., return the mean bucket value of the current subtree if the current subtree appears to be empty)
8. **Break**
9. **Else**
10. **Let** $t' := \Sigma_{y \in Y'} c_y$ be the sum of counters in Y'
11. **Let** $t'_z := \Sigma_{y \in Y', y \leq z} c_y$ be the partial sum of counters in Y' whose index is smaller than z
12. **Let** z' be child of lowest index for which $t'_{z'} \geq q \cdot t'$ (think of z' as the root of the subtree in which we expect the quantile to be)
13. **Set** $q := (q - (t'_{z'} - c_{z'}) / t') / (c_{z'} / t')$ (i.e., update q to mark the quantile relative to the new subtree at z')
14. **Set** $x := z'$
15. **Let** l_x be the lower bound of the domain of the subtree of x
16. **Let** r_x be the upper bound of the domain of the subtree of x
17. **Return** $(1 - q) \cdot l_x + q \cdot r_x$ (this is just a linear interpolation between l_x and r_x)

Note that all of this is post-processing on the differentially private counters, so the noisy quantile search is differentially private as well. To see a more detailed explanation, see the privacy proof.

Computational Performance

Quantile Trees can be built in a distributed fashion. The asymptotic time complexity of each operation as well as the space complexity is:

- **Insert:** $O(h)$, because we need to increment one counter per level of the tree
- **Merge:** $O(b^h)$, because we need to merge all nodes of the tree
- **Query for a Quantile:** $O(b \cdot h)$, because we need to look at/noise b nodes per layer when searching the quantile
- **Space per Tree:** $O(b^h)$, which is the number of nodes of a complete Quantile Tree

A more refined estimate of the merge operation can be obtained by taking the laziness into account. As a result, the runtime is linear in the number of non-empty nodes rather than the total number of nodes. This can result in a speed up if the input is sparse.

Because the runtime of a merge operation depends on the number of (non-empty) nodes, distributed computation will yield a meaningful speedup, if you can partition the input such that each partition is significantly larger than the number of nodes it takes up in the Quantile Tree.

Choosing the Height and Branching Factor

The height h and branching factor b of a Quantile Tree impact the performance and utility of the mechanism. For instance, the number of buckets into which the domain of the dataset is partitioned depends directly on h and b. However, these parameters also affect the amount of noise that needs to be added. Generally speaking, fewer buckets mean less noise but also less granular data. So, assuming you want to have a certain number n of buckets, how should you choose h and b?

To get some guidance and intuition, it makes sense to look at the sum of noise contributing to the outcome of a particular query. The noisy search algorithm queries at most $h \cdot b$ nodes, that is b nodes per layer. Moreover, assuming we are using Laplace noise, the standard deviation of the noise added to each node is in $O(h)$ (conveniently hiding the ε parameter in the asymptotic notation). According to the central limit theorem, the standard deviation of the sum of the noise is in $O(\sqrt{h \cdot b} \cdot h)$. Let's set $n = 1024$ and look at the implications of this based on three examples:

- **h=1, b=1024:** In this case, there is no tree structure at all. We are just looking at an array of buckets. When estimating the noise, we get a value of $\sqrt{1 \cdot 1024} \cdot 1 = 32$.
- **h=2, b=32:** In this setup the tree only has two layers and our noise estimate is $\sqrt{2 \cdot 32} \cdot 2 = 8$.
- **h=10, b=2:** This setup results in a full binary tree with estimated noise of $\sqrt{10 \cdot 2} \cdot 10 \approx 44.7$.

Based on this analysis, we would expect a tree of height 2 to perform best in this setup and the other two configurations significantly worse. Of course, these are only very rough approximations of the added noise (and there are other things to consider, such as introduced bias), but the general order of accuracy seems to be reflected in the experimental evaluation shown in Figure 2 below.

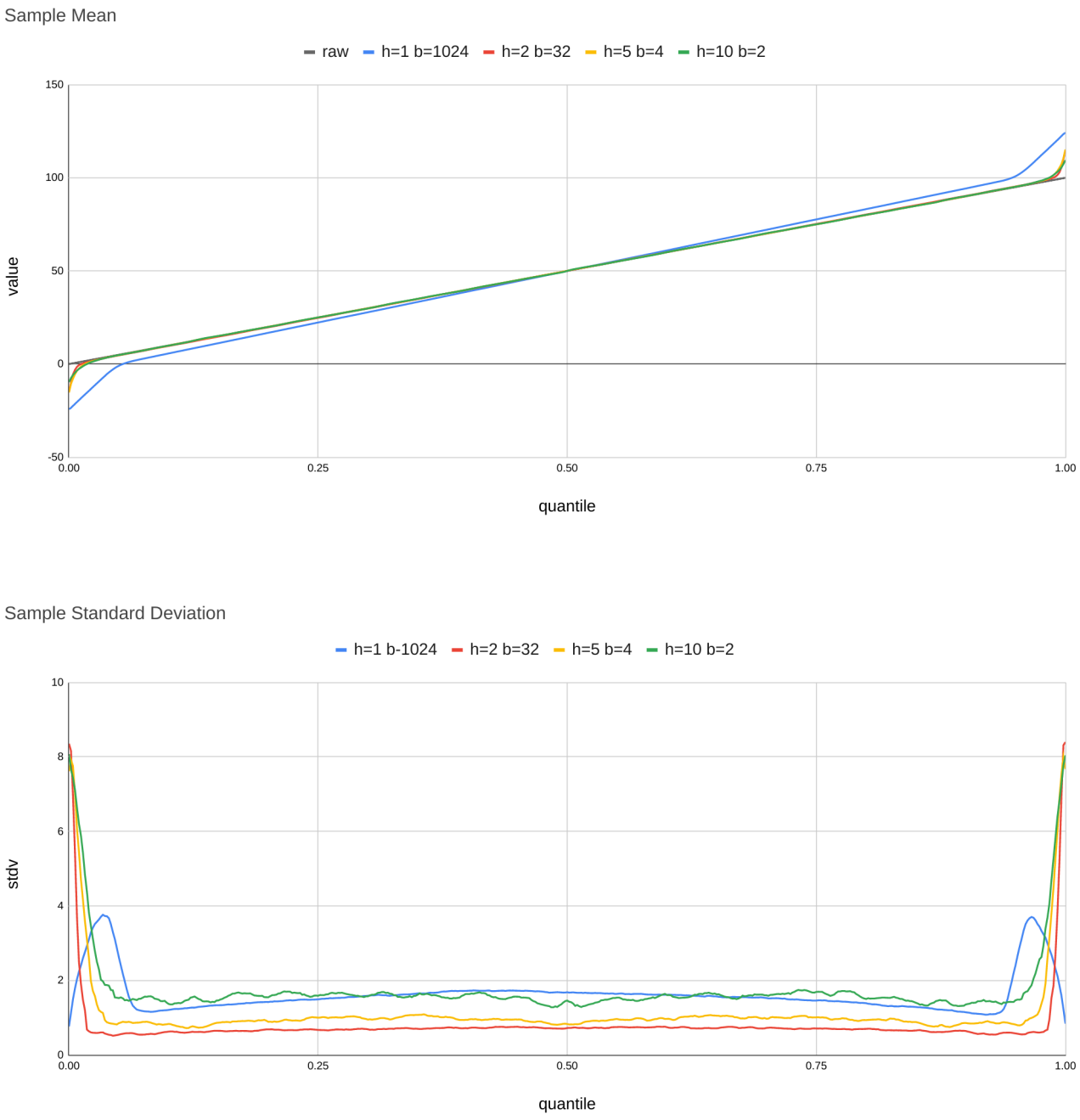

Figure 2: Statistical comparison of a Quantile Trees with different values of h and b on a linearly distributed dataset.

What you see in this figure is the sample mean and sample standard deviation of four different configurations of a Quantile Tree with **1024** buckets. Each configuration is sampled **1000** times per quantile. The dataset consists of **1000** equally spaced samples between **0** and **100**. Moreover, the lower and upper bounds are set to **-25** and **125** respectively. As a basic rule of thumb, it makes sense to keep the height of the tree small but greater than 1.

Proof of Privacy

The first thing to observe is that the noisy quantile search only looks at noised counters. So the output generated by the noisy quantile search on a lazy Quantile Tree is indistinguishable from the output the search would generate on a non-lazy Quantile Tree, i.e., a Quantile Tree where every counter is noised immediately upon entering the query state. Thus, if a non-lazy Quantile Tree is DP, so is the output of a lazy Quantile Tree (which is a different statement from claiming that the lazy data structure itself is DP, which is not the case because it stores the raw values).

Convincing yourself that a non-lazy Quantile tree is DP is easy. The key insight is that each time an individual contributes a piece of data to a Quantile Tree, at most h nodes of the tree are incremented by 1 (we can ignore the root node since it is not used for the quantile search and thus does not affect the result). Consequently, the **L1 Sensitivity** of the data structure is $h \cdot k$ where k is the maximum number of contributions a single individual can make to the dataset. If we add appropriately scaled Laplace or Gaussian noise to each node of the tree, the whole tree becomes DP q.e.d.