Logging Every Footstep:
Quantile Summaries for the Entire History*

Yufei Tao[1] Ke Yi[2] Cheng Sheng[1] Jian Pei[3] Feifei Li[4]

[1]Chinese University of Hong Kong
{*taoyf*, *csheng*}*@cse.cuhk.edu.hk*

[2]Hong Kong University of Science and Technology
yike@cse.ust.hk

[3]Simon Fraser University
jpei@cs.sfu.ca

[4]Florida State University
lifeifei@cs.fsu.edu

Abstract

Quantiles are a crucial type of order statistics in databases. Extensive research has been focused on maintaining a space-efficient structure for approximate quantile computation as the underlying dataset is updated. The existing solutions, however, are designed to support only the current, most-updated, snapshot of the dataset. Queries on the past versions of the data cannot be answered.

This paper studies the problem of *historical quantile search*. The objective is to enable ϵ-approximate quantile retrieval on *any* snapshot of the dataset in history. The problem is very important in analyzing the evolution of a distribution, monitoring the quality of services, query optimization in temporal databases, and so on. We present the first formal results in the literature. First, we prove a novel theoretical lower bound on the space cost of supporting ϵ-approximate historical quantile queries. The bound reveals the fundamental difference between answering quantile queries about the past and those about the present time. Second, we propose a structure for finding ϵ-approximate historical quantiles, and show that it consumes more space than the lower bound by only a square-logarithmic factor. Extensive experiments demonstrate that in practice our technique performs much better than predicted by theory. In particular, the quantiles it returns are remarkably more accurate than the theoretical precision guarantee.

---

*A preliminary version of the paper has appeared in SIGMOD'10. This version mainly differs in the proof of Theorem 2, which fixes a bug in the original proof.

1 Introduction

Quantiles are widely recognized as a crucial type of order statistics in databases. Specifically, let D be a set of N ordered data items. Given a parameter $\phi \in (0, 1]$, the *k-th ϕ-quantile* ($k = 1, 2, ..., \lfloor 1/\phi \rfloor$) is the item of D at the rank $\lfloor k\phi N \rfloor$ in the ordered sequence. An important problem is to maintain a space-efficient structure to support quantile retrieval when D is updated with insertions, and sometimes deletions as well. The problem has been extensively studied, as will be reviewed in Section 1.2. The focus of the relevant works, however, is to design a *snapshot* structure on the current D. Namely, after D is updated, the previous state of the structure is not preserved. As a result, quantile queries on the past versions of D cannot be answered.

In this paper, we consider the problem of *historical quantile retrieval*. Our goal is to support quantile queries on *all* the past snapshots of D. To illustrate, imagine that an initially empty D has been modified with an arbitrary sequence of insertions and deletions. In other words, D has evolved through a number of *versions*, one after each update. We aim at building a space-efficient structure that allows us to compute accurate ϕ-quantiles in any version of D in history.

Historical quantiles are useful for many purposes. Similar to "snapshot quantiles" that serve as a succinct summary of the data distribution in the current D, historical quantiles capture the entire *evolution* of the distribution of D. For example, in monitoring the performance of a web server, the server's present efficiency can be reflected by all the $\frac{1}{10}$-quantiles on the response times for the recent requests (i.e., the longest delay of the 10% fastest responses, of the 20% fastest, ...). Then, historical quantiles provide a clear picture of how the server's efficiency has been changing over time. Such information is vital to studying the server's usage and the behavior of users. Similar applications can be found in a variety of contexts involving different study subjects, such as the waiting time of a hotline, appeal processing time, elapsed time till the police's arrival after a 911 call, and so on.

Historical quantiles are also helpful to query optimization in *temporal databases*, which track the historical changes of a dataset (see [17] for an excellent survey). Consider, for instance, a temporal database that manages the balances of bank accounts. After a deposit/withdrawal, the previous balance of the affected account is not discarded, but instead, needs to be preserved in the database. The objective is to enable retrieval of the past data, such as "find all accounts whose balances were larger than 1 millon on 1 Jan. 2009". As with relational databases, effective query optimization in temporal databases also demands accurate estimation of the result sizes. Historical quantiles serve the purpose very well. In fact, it is well-known that quantiles are closely related to *range counting*: given an interval $[x_1, x_2]$, a range count query on D returns how many items of D fall in the interval. Typically, a structure for quantile computation can also be leveraged to perform range counting with good precision.

It is easy to see that, to answer historical quantile queries *exactly*, we must capture all the updates of D in history, which requires expensive space consumption. On the other hand, it has been well acknowledged that approximate quantiles already fulfill the purposes of many applications (in fact, most of the previous studies focus only on approximate quantiles). Informally, an approximate quantile returns items whose ranks only slightly differ from the desired ranks, by just a few percentage points. The benefit in return is that the amount of necessary space (for computing approximate historical quanties) can be significantly reduced.

This work presents the first formal results on approximate historical quantile retrieval. We propose a structure and its accompanying query algorithm to find historical quantiles with strong precision guarantees, which match the guarantees of the existing snapshot structures (i.e., the quantiles returned are always *ϵ-approximate*, as formalized shortly). Our structure is *deterministic*, namely, it always correctly answers all queries, instead of failing occasionally as in a probabilistic solution. Moreover, we prove a lower bound on how much space must be spent in the worst case by any structure, in order to support all (historical quantile) queries. The lower bound shows that the space cost of our structure is tight, up to only a square-logarithmic factor. Our theoretical results are verified by extensive experiments, which also demonstrate that, in practice,

the proposed structure works much better than predicted by theory. Specifically, the actual query results have errors that are significantly lower than the theoretical upper bounds.

The rest of the section will formally define the problem, review the previous results related to our work, and summarize our results. The subsequent sections are organized as follows. Section 2 proves the space lower bounds for ϵ-approximate historical quantile retrieval. Section 3 presents the proposed structure and studies its efficiency theoretically. Section 4 evaluates the practical performance of our technique with extensive experiments. Finally, Section 5 concludes the paper with directions for future work.

1.1 Problem definition

Let D be the dataset for the interest of quantile retrieval. We consider that the items in D are integers, but it is straightforward to apply our results to any ordered domain. Each integer is assumed to fit in a constant number of words.

D is initially empty. We are given the sequence of all updates on D in history. Denote by M the total number of updates. Each update is either an insertion or a deletion. Specifically, an *insertion* adds an integer to D, while a *deletion* removes an existing number in D. Denote by $D(i)$ the snapshot of D after the i-th update ($1 \leq i \leq M$).

We will refer to $D(i)$ as the *version* i of D. Let $N(i)$ be the size of $D(i)$, namely, $N(i) = |D(i)|$. Obviously, since an insertion (deletion) increases (decreases) the size of D by 1, it follows that $|N(i+1) - N(i)| = 1$. We refer to the ordered list $(N(1), ..., N(M))$ as the *size sequence*. Without loss of generality, we assume $N(i) > 0$ for all $1 \leq i \leq M$. Otherwise, the update sequence can be broken into several continuous segments, each of which satisfies the assumption, and can be processed separately.

Given a parameter $\phi \in (0, 1]$, the *ϕ-quantile* of $D(i)$ is the $\lfloor \phi N(i) \rfloor$-th greatest[1] item in $D(i)$. Alternatively, this is the item with *rank* $\lfloor \phi N(i) \rfloor$ in non-ascending order of the items in $D(i)$. Ties are broken arbitrarily. A related concept, which appeared in some previous works, is the *k-th ϕ-quantile* ($1 \leq k \leq \lfloor 1/\phi \rfloor$), which in our context is the item with rank $\lfloor k\phi N(i) \rfloor$ in $D(i)$. For the purpose of our discussion, it suffices to regard the k-th ϕ-quantile simply as the $k\phi$-quantile.

We aim at retrieving *ϵ-approximate quantiles*. Formally, given $\epsilon \in (0, 1]$, an *ϵ-approximate ϕ-quantile* of $D(i)$ is a value u in the data domain fulfilling two conditions:

- At least $(\phi - \epsilon)N(i)$ items of $D(i)$ are greater than or equal to u;
- At most $(\phi + \epsilon)N(i)$ items of $D(i)$ are greater than or equal to u.

Intuitively, these conditions imply that the rank of u differs from the requested rank $\phi N(i)$ by at most $\epsilon N(i)$. Usually multiple values can be returned as the result u. Precisely, u can be any value that is not smaller than the item in $N(i)$ with rank $\lceil (\phi - \epsilon)N(i) \rceil$, and strictly smaller than the item in $N(i)$ with rank $\lfloor (\phi + \epsilon)N(i) \rfloor + 1$

Given a value of ϵ, our goal is to pre-process the historical updates of D into a structure that can be used to answer ϵ-approximate quantile queries with any $\phi \in (0, 1]$, in all possible versions $i \in [1, M]$ of D. The structure should consume small space, but needs to answer each query efficiently, both in the worst case.

1.2 Previous work

To our knowledge, no formal result is known in the literature for computing historical quantiles. The previous research mainly focused on approximate quantile search in the current, most updated, snapshot of the dataset

[1] By symmetry, our technique can be easily adapted in case the ϕ-quantile is defined as the $\lfloor \phi N(i) \rfloor$-th *smallest* item in $D(i)$.

D. The objective is to maintain a structure along with the updates on D, so that it can be used to answer queries correctly. The challenge is to minimize the space of the structure.

Most works consider that D is updated with only insertions, i.e., no item in D is ever removed. In this setting, Munro and Paterson [16] suggested a structure for ϵ-approximate quantile retrieval that consumes $O(\frac{1}{\epsilon}\log^2(\epsilon N))$ space, where N is the number of insertions. Various heuristic improvements (without affecting the space complexity) were discussed and experimented in [1, 14]. Manku et al. [15] proposed a randomized structure that successfully answers a query with probability at least $1-\delta$ for any $\delta \in (0,1]$, and occupies $O(\frac{1}{\epsilon}(\log^2\frac{1}{\epsilon} + \log^2\log\frac{1}{\delta}))$ space. Greenwald and Khanna [10] gave a deterministic structure that requires $O(\frac{1}{\epsilon}\log(\epsilon N))$ space. Cormode et al. [5] extended the structure of [10] to compute *biased quantiles*, which were also studied by Gupta and Zane [11].

ϵ-approximate quantile search is much harder when deletions are allowed. Only a few results exist in this setting. Gilbert et al. [9] are the first to tackle this challenge. They designed a randomized structure that requires $O(\frac{1}{\epsilon^2}\log^2 U \log\frac{\log U}{\delta})$ space, where $1-\delta$ is the success probability of answering a query, and U is the size of the data domain. The *count-min* sketch of Cormode and Muthukrishnan [6] reduces the space by a factor of $1/\epsilon$. In the *sliding-window model*, where D contains only the R items most recently received, Arasu and Manku [2] proposed a structure requiring $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon}\log R)$ space. Note, however, that the sliding window model imposes a strict ordering on the insertions and deletions. Hence, the solution of [2] cannot be applied to sequences of updates where insertions and deletions are arbitrarily mixed.

As for *exact* quantiles, Blum et al. [4] were the first to show that the median (i.e., the $\frac{1}{2}$-quantile) of a dataset with N items can be found in $O(N)$ time. Their algorithm can be modified to retrieve any ϕ-quantile with the same time bound.

It would be tempting to adapt one of the above methods to our historical quantile problem, but the adaption seems to be either impossible or space expensive. First, obviously only methods that support *arbitrary* deletions can be considered, which already eliminates all the approximate solutions, except [6, 9]. Unfortunately, the algorithms of [6, 9] are quite specific to their own settings; it is not clear how their footprints in history can be compacted effectively for querying historical quanties. Naive extensions result in $\Omega(M)$ storage. In any case, the solutions of [6, 9] are probabilistic (i.e., they may fail occasionally), as opposed to our interest in deterministic algorithms. Finally, although the exact algorithm of [4] can be directly applied to find a historical quantile, its space and query costs $O(M)$ are excessive for our goal.

1.3 Our main results

The first major contribution of this paper is a space lower bound of any (deterministic or random) structure for ϵ-approximate quantile search in history. The lower bound is related to ϵ, the number M of updates, and somewhat surprisingly, the *Harmonic mean* H of the sizes of D at all past versions. Specifically, as defined in Section 1.1, let $N(i)$ be the size of the i-th version of D, $1 \le i \le M$; then, H is given by:

$$H = \frac{M}{\sum_{i=1}^{M}\frac{1}{N(i)}}. \tag{1}$$

We show that any structure must consume at least $\Omega(\frac{M}{\epsilon H})$ space in the worst case to answer all queries correctly. This bound is drastically different from the well-known space lower bound $\Omega(1/\epsilon)$ for computing *snapshot* ϵ-approximate quantiles. In particular, the presence of H in our bounds is a unique feature of historical quantile retrieval.

The paper's second contribution is a new deterministic structure for answering ϵ-approximate historical quantile queries. The structure requires $O(\frac{M}{\epsilon H}\log^2\frac{1}{\epsilon})$ space, which is higher than the theoretical lower bound by only a factor of $O(\log^2\frac{1}{\epsilon})$. Its accompanying query algorithm takes $O(\log\frac{1}{\epsilon} + \log\frac{M}{H})$ time to compute an ϵ-approximate quantile. Note that none of our bounds depends on the size U of the data domain.

Our experiments deployed both synthetic and real data to evaluate the efficiency of our techniques. The results indicate that, by consuming *only 1%* of the underlying dataset, the proposed structure can be used to find very accurate quantiles throughout the entire history. Furthermore, the accuracy increases gracefully when additional space is allowed.

2 Space lower bounds for historical quantile search

In this section, we will derive a lower bound on the space needed to support ϵ-approximate historical quantile queries. First of all, it is easy to see that, in some extreme scenarios, the space cost must be $\Omega(M)$, where M is the number of updates in history. For example, suppose that each past version $D(i)$ of the dataset D ($1 \leq i \leq M$) is so small that $N(i) = |D(i)| < \frac{1}{\epsilon}$. In this case, the error permitted by an ϵ-approximate quantile is less than $\epsilon N(i) < 1$. In other words, no error is allowed, and we must return the *exact* quantiles. This, in turn, means that all the updates in history must be captured, which requires $\Omega(M)$ space.

The situation in practice is much better, because typically $N(i)$ is by far larger than $1/\epsilon$, which makes it possible to use much less space than $\Omega(M)$ to find ϵ-approximate historical quantiles. Nevertheless, the above discussion implies that the size of each past version of D should play a role in the minimum amount of space needed. It turns out that there is a close relationship between the space and the Harmonic mean H of the sizes of all the versions of D (see Equation 1).

In the sequel, we will establish the relationship in two steps. First, we give a lower bound that is weaker than our final lower bound, but its proof is simpler, and illustrates the main ingredients of our methodology. The analysis will then be extended to obtain our ultimate lower bound.

The first lower bound. The bound is stated in the following theorem:

Theorem 1. *For any size sequence* $(N(1), N(2), ..., N(M))$ *and* $\epsilon \leq 1/3$*, a structure for answering historical* ϵ*-approximate quantile queries must use* $\Omega(M/H)$ *words in the worst case.*

Our earlier discussion already demonstrates that the theorem is correct if $N(i) < 1/\epsilon$ for all $1 \leq i \leq M$, noticing that the H of such a size sequence is less than $1/\epsilon$, so $\Omega(M/H) = \Omega(M)$. Next, we will show that the theorem is correct for *any* size sequence. Towards this purpose, we first prove a relevant lemma:

Lemma 1. *For any* $\epsilon < 1/3$ *and any size sequence* $(N(1), N(2), ..., N(M))$*, there exists a sequence of updates such that, if we query the* ϵ*-approximate median at each version* $i \in [1, M]$*, the number of different results (returned by any structure) must be* $\Omega(M/H)$*.*

To prove the lemma, we construct a hard sequence of updates, based on the following principle: *an insertion always inserts a value greater than the maximum value in D, and a deletion always removes the minimum value in D*. Whether the i-th ($i \geq 1$) update is an insertion or deletion depends on the comparison of $N(i)$ and $N(i-1)$ (recall that $|N(i) - N(i-1)| = 1$). For example, assume $M = 6$, and that the size sequence is $(1, 2, 3, 2, 3, 2)$. The constructed sequence of updates can be $(+10, +20, +30, -10, +40, -20)$, where $+u$ ($-u$) represents an insertion (deletion) of value u. Note that each inserted value is greater than all the previously inserted values, whereas each deletion removes the smallest value that has not been deleted.

Next, we define a sequence of values $(v(1), v(2), ...)$. They indicate some special versions of D that are important to space consumption, as elaborated later. Specifically:

- $v(1) = 1$.
- Inductively, $v(i+1)$ is the smallest $j > v(i)$ satisfying *either* of the following conditions:

  $C1$: $N(j) \geq N(v(i))/(1/2 - \epsilon)$;

$C2$: there have been $N(v(i)) \cdot (1/2 + \epsilon)$ deletions between versions $v(i)$ and j.

Intuitively, Condition $C1$ means that the size of D at version $v(i+1)$ has increased by a factor of $\frac{1}{1/2-\epsilon}$ compared to the size at version $v(i)$. Condition $C2$, on the other hand, indicates that many deletions have occurred between versions $v(i)$ and $v(i+1)$. The inductive definition of $v(i+1)$ continues until such a j cannot be found. Let L be the number of values in the sequence, i.e., $v(1)$, $v(2)$, ..., $v(L)$ are successfully defined. The following is a useful property about these values:

Lemma 2. $v(i+1) - v(i) \leq N(v(i)) \cdot \frac{3-4\epsilon^2}{1-2\epsilon}$ *for any* $1 \leq i \leq L-1$.

Proof. Note that $v(i+1) - v(i)$ equals the number of updates to transform version $v(i)$ to $v(i+1)$. Among them, there can be at most $N(v(i)) \cdot (1/2 + \epsilon)$ deletions due to Condition $C2$. Thus, before Condition $C1$ is violated, there can be at most $\frac{N(v(i))}{1/2-\epsilon} + N(v(i)) \cdot (1/2+\epsilon)$ insertions. This makes the total number of updates at most $\frac{N(v(i))}{1/2-\epsilon} + 2N(v(i)) \cdot (1/2+\epsilon) = N(v(i)) \cdot \frac{3-4\epsilon^2}{1-2\epsilon}$. □

Recall that, as mentioned in Section 1.1, there may be more than one value that can be returned as an ϵ-approximate ϕ-quantile. For proving Lemma 1, we consider only $\phi = 1/2$, namely, the ϕ-quantile is the median. For the i-th version $D(i)$ of D, $1 \leq i \leq M$, we define a *legal range* to be the range of values (in the data domain) that can be an ϵ-approximate median of $D(i)$. More precisely, the legal range starts from the item of $D(i)$ at rank $\lceil (1/2 - \epsilon)N(i) \rceil$, and ends right before (but does not touch) the item at rank $\lfloor (1/2 + \epsilon)N(i) \rfloor + 1$. Now we give a fact which reveals why a certain amount of space is inevitable:

Lemma 3. *The legal ranges at versions* $v(1)$, $v(2)$, ..., $v(L)$ *are mutually disjoint.*

Proof. We will show that the legal range of $v(i+1)$ must be completely above $v(i)$ for each $i \geq 1$. We distinguish two cases, depending on how j is decided in defining $v(i+1)$.

Case 1: j is decided by Condition $C1$. It follows that $N(v(i+1)) \geq \frac{N(v(i))}{1/2-\epsilon}$. Let cut be the maximum value in D at version $v(i)$. Clearly, cut has rank 1 in $D(v(i))$, which is lower than $N(v(i))/2$ by more than $\epsilon N(v(i))$ (due to $\epsilon < 1/2$)). Hence, the legal range of version $v(i)$ must finish strictly before cut. On the other hand, there are $N(v(i+1)) - N(v(i))$ items in $D(v(i+1))$ greater than cut. From $N(v(i+1)) \geq \frac{N(v(i))}{1/2-\epsilon}$, we know

$$
\begin{aligned}
N(v(i+1)) - N(v(i)) &\geq N(i+1)(1 - (1/2 - \epsilon)) \\
&= N(i+1)(1/2 + \epsilon).
\end{aligned}
$$

Therefore, the legal range of $v(i+1)$ must start strictly after cut.

Case 2: j is decided by Condition $C2$. Let cut be the minimum value in D at version $v(i+1)$. Notice that the deletions during the period from $v(i)$ to $v(i+1)$ remove all the items of $D(i)$ below cut. With reasoning similar to Case 1, it is easy to verify that the legal range of version $v(i)$ must finish strictly before cut, while that of $v(i+1)$ must start strictly after cut. □

The above lemma implies that no common value can be returned as an ϵ-approximate median simultaneously for any two of the versions $v(1)$, $v(2)$, ..., $v(L)$. Equivalently, any structure must return L different values to be the ϵ-approximate medians at those L versions, respectively.

Next, we complete the proof of Lemma 1 by showing $L \geq \frac{(1-2\epsilon)^2}{6-8\epsilon^2}\frac{M}{H} = \Omega(M/H)$. For this purpose, define $v(L+1) = M+1$ to tackle the boundary case in the following equation:

$$\sum_{i=1}^{M} \frac{1}{N(i)} = \sum_{i=1}^{L} \sum_{j=v(i)}^{v(i+1)-1} \frac{1}{N(j)}.$$

By Condition $C2$, for any $j \in [v(i), v(i+1)-1]$, $N(j) \geq N(v(i)) - N(v(i)) \cdot (1/2+\epsilon) = N(v(i)) \cdot (1/2-\epsilon)$. Thus:

$$\begin{aligned}
\sum_{j=v(i)}^{v(i+1)-1} \frac{1}{N(j)} &\leq \sum_{j=v(i)}^{v(i+1)-1} \frac{1}{N(v(i)) \cdot (1/2-\epsilon)} \\
&= \frac{v(i+1)-v(i)}{N(v(i)) \cdot (1/2-\epsilon)} \\
&\leq \frac{N(v(i)) \cdot \frac{3-4\epsilon^2}{1-2\epsilon}}{N(v(i)) \cdot (1/2-\epsilon)} \quad \text{(By Lemma 2)} \\
&= \frac{6-8\epsilon^2}{(1-2\epsilon)^2}.
\end{aligned}$$

Therefore:

$$\frac{M}{H} = \sum_{i=1}^{M} \frac{1}{N(i)} \leq \frac{6-8\epsilon^2}{(1-2\epsilon)^2} L.$$

Hence, by the fact $\epsilon \leq 1/3$, we know

$$L \geq \frac{(1-2\epsilon)^2}{6-8\epsilon^2} \cdot \frac{M}{H} \geq \frac{(1-2/3)^2}{6} \cdot \frac{M}{H}$$

which is what we need for Lemma 1.

Using a standard information theoretical argument, we can carefully specify the inserted values (each fitting in $O(1)$ words) such that there are at least $2^{\Omega(MW/H)}$ possibilities for the set of results at all $v(i)$, $1 \leq i \leq L$, where W is the length of a word. This establishes Theorem 1.

A tighter lower bound. The above analysis can be extended to obtain a stronger result. For simplicity, in the rest of this section, we will focus on the situation where $1/\epsilon$ is an integer.

Theorem 2. *For any $\epsilon \leq 1/4$ and any size sequence $(N(1), N(2), ..., N(M))$ satisfying*

- $N(i) = i$ *for* $i = 1, 2, ..., 1/\epsilon$
- $N(i) \geq 1/\epsilon$ *for all* $i \in [1+1/\epsilon, M]$

a structure for answering historical ϵ-approximate quantile queries must use $\Omega(\frac{1}{\epsilon}(\frac{M}{H} - \ln\frac{1}{\epsilon} - \Theta(1)))$ words in the worst case.

To prove Theorem 2, we deploy the same hard update sequence of D that was used earlier to prove Theorem 1. Let us first re-define $v(1)$, $v(2)$, ... in a slightly different way:

- $v(1) = 1/\epsilon$.

- Inductively, $v(i+1)$ is the smallest $j > v(i)$ satisfying *any* of the following conditions:

  $C1$: $N(j) \geq N(v(i))/(1/2 - \epsilon)$;

  $C2.1$: there have been $N(v(i)) \cdot (1/2+\epsilon)$ deletions between versions $v(i)$ and j; furthermore, $N(v(j)) \geq \frac{3}{4}N(v(i))$;

  $C2.2$: same as $C2.1$ but $N(v(j)) < \frac{3}{4}N(v(i))$.

The inductive definition continues until j is undefined. Denote by L the total number of values defined.

Observe that when Condition $C1$ or $C2.1$ is satisfied, at least $N(v(i))/4 = \Omega(N(v(i))$ new items have been added to D after version $v(i)$. Moreover, *all* these items rank ahead of the items of $D(v(i))$. Using a standard argument to design the inserted values, we can show that, after version $v(i)$, $\Omega(1/\epsilon)$ extra words are needed to support ϵ-approximate queries at version $v(i+1)$ on the ranks of those items inserted after $v(i)$.

There is another imperative observation. Let x_1, x_2, and x_3 be the number of times Conditions $C1$, $C2.1$, and $C2.2$ are satisfied, respectively. It must hold that $x_3 = O(x_1 + x_2)$. This is because

- at each occurrence of $C1$ or $C2.1$, the cardinality of D can be at most $\frac{1}{1/2-\epsilon} \leq 4$ times that of $D(v(i))$, and

- at each $C2.2$, the cardinality D must be smaller than that of $D(v(i))$ by a factor of at least $3/4$.

Therefore:

$$N(1/\epsilon) \cdot 4^{x_1+x_2} \cdot (3/4)^{x_3} \geq N(M)$$

which, with the fact $N(M) \geq 1/\epsilon = N(1/\epsilon)$, leads to

$$x_3 \leq \frac{\log 4}{\log(4/3)}(x_1 + x_2) = O(x_1 + x_2).$$

Using the argument proving Lemma 1, we know that $L = x_1 + x_2 + x_3 = \Omega(M'/H')$, where $M' = M - (1/\epsilon) + 1$ (i.e., excluding the first $(1/\epsilon) - 1$ updates), and H' is the Harmonic mean of $N(1/\epsilon), N(1 + 1/\epsilon), ..., N(M)$. Hence, $x_1 + x_2 = \Omega(M'/H')$.

Observe that

$$\begin{aligned} \frac{M}{H} &= \sum_{i=1}^{M} \frac{1}{N(i)} = \sum_{i=1}^{(1/\epsilon)-1} \frac{1}{N(i)} + \sum_{i=1/\epsilon}^{M} \frac{1}{N(i)} \\ &\leq \ln(1/\epsilon) + \Theta(1) + \frac{M'}{H'} \end{aligned}$$

Hence, $L = \Omega(M'/H') = \Omega(\frac{M}{H} - \ln(1/\epsilon) - \Theta(1))$. Hence, $x_1 + x_2 = \Omega(\frac{M}{H} - \ln(1/\epsilon) - \Theta(1))$. In other words, $\Omega(1/\epsilon)$ new words must be stored at each of at least $\Omega(\frac{M}{H} - \ln(1/\epsilon) - \Theta(1))$ versions, which completes the proof of Theorem 2.

Remark 1. Theorem 2 suggests that the space must be $\Omega(\frac{M}{\epsilon H})$ when $M/H \geq 2\ln(1/\epsilon)$.

Remark 2. An interesting special case is when all the M updates are insertions. In this case, the $D(i)$ has exactly $N(i) = i$ items. As a result, $M/H = \sum_{i=1}^{M} 1/i \leq \ln M$. So the space lower bound becomes $\Omega(\frac{1}{\epsilon}(\ln \epsilon M - \Theta(1))$.

3 A structure for historical quantile search

Section 3.1 briefly reviews the *persistence technique* since it is deployed by our solutions. Section 3.2 explains the high-level ideas behind the proposed structure, which is formally described in Section 3.3, together with its query algorithm. Section 3.4 elaborates how to construct our structure, and Section 3.5 analyzes its space and query complexities. Finally, in Section 3.6, we give another solution that is fairly simple to implement.

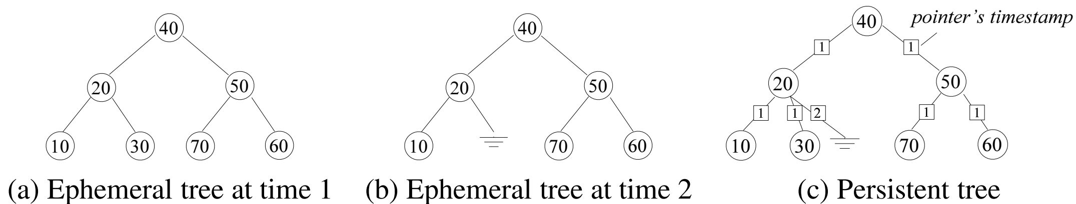
3.1 Persistence technique

The *persistence framework* [8] (also known as the *multi-version framework* [3]) is a general technique for capturing all the historical changes of a dynamic structure. In the database area, it has been applied to design many access methods (see, for example, [3, 7, 13, 18, 19]). To illustrate, assume that the underlying structure is a binary tree $\mathcal{T}$ (e.g., a *red-black tree* [8]). Each node n of $\mathcal{T}$ stores an index key k_n and a constant-size information tag t_n. We consider three update operations on $\mathcal{T}$:

1. *insert*(k, t)*:* insert a node with key k and information tag t;
2. *delete*(n)*:* delete a node n;
3. *modify*(n, t)*:* reset the information tag t_n of node n to t.

Traditionally, a binary tree is *ephemeral* because, after an update, the previous version of $\mathcal{T}$ is lost. A *persistent binary tree* $\mathcal{T}^p$ [8], on the other hand, retains all the past versions of $\mathcal{T}$ (one version per update) in a space-efficient manner.

For example, consider a binary tree at time 1 as shown in Figure 1a. At time 2, key 30 is deleted, and the tree changes to Figure 1b. The corresponding persistent tree is demonstrated in Figure 1c. Each pointer is associated with its creation *timestamp* (note that a timestamp is stored with an edge, as opposed to the information tag in a node). The purpose of such timestamps is to identify the ephemeral binary tree of a specific version. For example, assume that we want to single out, from the tree in Figure 1c, the nodes of the ephemeral tree at time 2. Although node 20 has two right pointers, no ambiguity can be caused: clearly, the one carrying timestamp 2 belongs to the tree at time 2.



(a) Ephemeral tree at time 1 (b) Ephemeral tree at time 2 (c) Persistent tree

Figure 1: Illustration of the persistent binary tree

As shown in [8], a persistent tree occupies $O(N)$ space, where N is the total number of updates (i.e., *insert*, *delete*, and *modify*) in history. In other words, each update incurs only $O(1)$ space. Finally, note that the persistence technique is a *framework*. That is, given a sequence of updates on a traditional binary tree $\mathcal{T}$, the corresponding persistent tree $\mathcal{T}^p$ can be created by standard algorithms [3, 8]. Hence, we can focus on explaining how to update $\mathcal{T}$ itself, without worrying about the technical details of maintaining $\mathcal{T}^p$ (including its edges' timestamps).

3.2 High-level rationales and challenges

The proposed technique is motivated by the fact that a binary tree can be used to query exact quantiles efficiently. To illustrate, assume that the dataset D consists of 10 integers: 10, 20, ..., 100. Figure 2 shows a binary tree on these values. Each node n carries an *r-counter* (as the information tag of n) that equals the number of values in the right subtree of n. For example, the r-counter of the root is 6 since its right subtree has 6 values 50, 60, ..., 100.

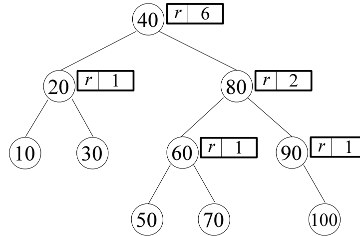


Figure 2: Using a binary tree to query quantiles

The binary tree allows us to find any exact ϕ-quantile by accessing *at most* a single path from the root to a leaf. For instance, assume $\phi = 1/2$, namely, the ϕ-quantile is the $\phi|D| = 5$-th greatest item in D. We can find such an item as follows. At any time, our algorithm maintains r_{total}, which intuitively is the number of items in D that are confirmed to be greater than the node being processed. This value is increased whenever we access the left child of a node. Initially, $r_{total} = 0$ and the first node visited is the root 40. Its r-counter 6 indicates that the result item (i.e., the 5-th greatest in D) must lie in the root's right subtree. Hence, we access its right child 80. The r-counter 2 of node 80 shows that the result item is in its left subtree. Before descending, however, we increase r_{total} by $2 + 1 = 3$, to indicate that 3 items are definitely greater than the node 60 we are about to process. Specifically, the 2 comes from the r-counter of 80, and the 1 refers to node 80 itself. Now, we visit node 60, and obtain its r-counter 1. Combined with $r_{total} = 3$, it is clear that exactly 4 items in D are greater than 60. Therefore, the algorithm returns 60 and terminates.

Extending the above idea, we could maintain such a binary tree on every snapshot of D in history, which would enable *exact* quantile queries on any snapshot. All these trees, however, require expensive space, but since our goal is ϵ-approximate quantile search, the space consumption can be reduced by permitting several types of imprecision in each tree. First, it is not necessary to create a node for every item in D, but instead, multiple items can be collectively represented by one node, which corresponds to an interval in the data domain. Second, the r-counters do not have to be fully accurate. Third, as opposed to using the exact $|D|$ in the above algorithm, we can work with an approximate $|D|$, to avoid keeping track of the exact size of every past snapshot of D. The challenge, however, is to develop all the above rationales into a concrete structure that can guarantee ϵ-approximate quantiles, and at the same time, consume small space.

3.3 The structure and its query algorithm

This subsection will formally discuss the proposed structure and its query algorithm. Denote by $\mathbb{D}$ the data domain. As before, let $D(i)$ be the i-th version of D (i.e., the snapshot of D after the i-th update), and $N(i)$ be the size of $D(i)$, $1 \leq i \leq M$. H is the Harmonic mean of $N(1)$, $N(2)$, ..., $N(M)$, as given in Equation 1.

Overview. Our structure is a forest of persistent binary trees $\mathcal{T}_1^p$, $\mathcal{T}_2^p$, ... each of which supports ϵ-approximate quantile search in some versions of D. Specifically:

1. $\mathcal{T}_1^p$ supports queries on $D(1)$ and $D(2)$.

2. Inductively, if $\mathcal{T}_j^p$ covers up to version $v-1$, then $\mathcal{T}_{j+1}^p$ supports queries on the next $1+\lfloor N(v)/2 \rfloor$ versions, namely, $D(v)$, $D(v+1)$, ..., $D(v+\lfloor N(v)/2 \rfloor)$.

The above construction continues until all the M versions of D have been covered.

Let T be the total number of persistent trees built. In Section 3.5, we will show that $T = O(M/H)$, and each tree occupies $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$ space. Hence, the total space cost is $O(\frac{M}{\epsilon H}\log^2\frac{1}{\epsilon})$.

Structure and properties. Next, we explain the details of each persistent tree $\mathcal{T}_j^p$, $1 \le j \le T$. Since all $\mathcal{T}_j^p$ have the same structure, we will drop the subscript j when there is no ambiguity. Furthermore, if $D(v)$ is the first version of D covered by $\mathcal{T}^p$, we say that v is the *initial version* of $\mathcal{T}^p$.

As explained in Section 3.1, a persistent binary tree can be understood as a set of ephemeral binary trees. As for $\mathcal{T}^p$, it captures $1+\lfloor N(v)/2 \rfloor$ ephemeral trees, that is, $\mathcal{T}(i)$ for $v \le i \le v+\lfloor N(v)/2 \rfloor$, where v is the initial version of $\mathcal{T}^p$. Each $\mathcal{T}(i)$ is used to answer queries on the i-th version $D(i)$ of D. It is similar to the binary tree in Figure 2, but permits some imprecision (mentioned in Section 3.2). Specifically:

- The key k_n of each node n in $\mathcal{T}(i)$ is an interval in the data domain $\mathbb{D}$, such that the keys of all nodes in $\mathcal{T}(i)$ constitute a *disjoint* partitioning of $\mathbb{D}$. $\mathcal{T}(i)$ is built on the natural ordering of these intervals.
- Each node n carries an information tag $t_n = (c_n, r_n)$, where
  1. c_n is (approximately) the number of items of $D(i)$ covered by the interval k_n of n. We refer to c_n as the *c-counter* of n.
  2. r_n is (approximately) the number of items of $D(i)$ covered by the intervals in the *right* subtree of n. It is called the *r-counter* of n. If n does not have a right child, r_n must be 0.
- The root $\hat{n}$ of $\mathcal{T}(i)$ has an extra information tag $ALL_{\hat{n}}$, which is (approximately) the size $N(i)$ of $D(i)$. We refer to $ALL_{\hat{n}}$ as the *ALL-counter* of $\hat{n}$.

Furthermore, four properties are ensured on $\mathcal{T}(i)$:

$P1$: There are $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})$ nodes[2] in $\mathcal{T}(i)$, whose height is therefore at most:

$$h = \alpha \log(1/\epsilon) \tag{2}$$

for a properly chosen constant α.

$P2$: $c_n \le \lceil B \rceil$, where

$$B = \epsilon N(v)/16 \tag{3}$$

with v being the initial version of $\mathcal{T}^p$.

$P3$: All the c- and r-counters can have errors at most:

$$E = \max\{0, B/h - 1\}. \tag{4}$$

In other words, let c_n^* be the accurate number of items in $D(i)$ covered by interval k_n, and let r_n^* be the accurate number of items in $D(i)$ covered by the intervals in the right subtree of n. Then, $|c_n - c_n^*| \le E$ and $|r_n - r_n^*| \le E$.

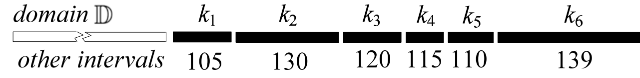
$P4$: For the root $\hat{n}$, $ALL_{\hat{n}}$ can have an error at most $\epsilon N(v)/4$, namely, $|ALL_{\hat{n}} - N(i)| \le \epsilon N(v)/4$.

[2]We can reduce the number of nodes in $\mathcal{T}(i)$ to $O(1/\epsilon)$, using a somewhat more complex construction algorithm than the one presented later. Details are left to the full paper.

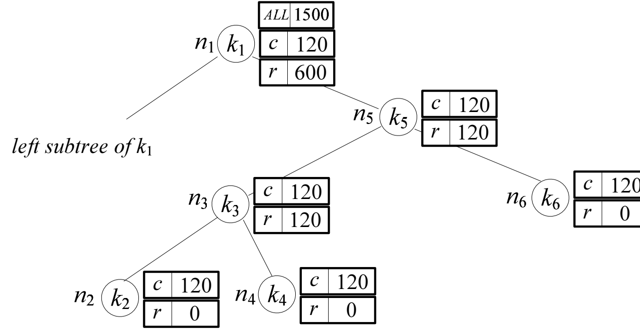
Example 1. Next, we give an example ephemeral binary tree $\mathcal{T}(i)$, assuming $B = 120$ and $E = 19$. Since the value of i is not important, we abbreviate $\mathcal{T}(i)$ as $\mathcal{T}$, and $D(i)$ as D.

Recall that the key of each node in $\mathcal{T}$ is an interval, and the keys of all nodes in $\mathcal{T}$ partition the data domain $\mathbb{D}$. Figure 3a shows the right-most few intervals k_1, k_2, ..., k_6 indexed by $\mathcal{T}$ (these intervals are the "largest" elements in $\mathcal{T}$). The number below each interval indicates how many items of D are covered by the interval (e.g., k_1 covers 105 items).

Figure 3b demonstrates part of $\mathcal{T}$. Observe that the nodes' counters are not all accurate, but their errors are at most $E = 19$. Consider, for example, the root; its interval k_1 covers 105 items, so its c-counter 120 has an error of 15. On the other hand, its r counter 600 has an error of 14. To see this, notice that the root's right subtree has 5 intervals k_2, k_3, ..., k_6. They cover totally $130 + 120 + ... + 139 = 614$ items (Figure 3a), which is 14 greater than the root's r-counter. Finally, note that the errors of counters can be either positive or negative. □



(a) Number of items of D covered by each interval



(b) Ephemeral binary tree $\mathcal{T}$

Figure 3: The structure of an ephemeral binary tree

Query algorithm. Before explaining how to build our structure, let us first give the algorithm for finding an ϵ-approximate historical quantile, as it will clarify many rationales behind our design. Assume that a query requests an ϵ-approximate ϕ-quantile in version $D(q)$. We answer it using the ephemeral binary tree $\mathcal{T}(q)$ on $D(q)$. Let $\lambda = \phi \cdot ALL_{\hat{n}}$, where $\hat{n}$ is the root of $\mathcal{T}(q)$.

Our algorithm accesses at most a single path of $\mathcal{T}(q)$. It maintains a value r_{total}, which equals the sum of the c- and r-counters of every node where we descended to its left child. Initially, $r_{total} = 0$, and the first node visited is the root. In general, after loading a node n, we proceed differently depending on the comparison between λ and the range $[r_{total} + r_n, r_{total} + c_n + r_n]$:

1. If $r_{total} + r_n \leq \lambda \leq r_{total} + c_n + r_n$, then terminate by returning the starting value of the key k_n of n (remember that k_n is an interval).

2. If $\lambda < r_{total} + r_n$, recursively visit the right child of n.

3. Otherwise (i.e., $\lambda > r_{total} + c_n + r_n$), first increase r_{total} by $c_n + r_n$, and then recursively visit the left child of n. If n does not have a left child, terminate by returning the starting value of k_n.

The query algorithm is formally presented in Figure 4.

algorithm *historical-quantile* (ϕ, q)

/* find an ϵ-approximate ϕ-quantile in version $D(q)$ */

1. $\mathcal{T}$ = the ephemeral binary tree for $D(q)$
2. $\lambda = \phi \cdot ALL_{\hat{n}}$, where $\hat{n}$ is the root of $\mathcal{T}$
3. $r_{total} = 0$; $n = \hat{n}$
4. **while** (true)
5.   **if** $r_{total} + r_n \leq \lambda \leq r_{total} + c_n + r_n$ **then**
6.   **return** the starting value of k_n
7.   **else if** $\lambda < r_{total} + r_n$ **then**
8. n = the right child of n
9.   **else**
10. $r_{total} = r_{total} + c_n + r_n$
11.   **if** n has a left child
12. n = the left child of n
13.   **else return** the starting value of k_n

Figure 4: Algorithm for finding historical quantiles

Example 2. To demonstrate the algorithm, assume that a query requests the $(4/15)$-quantile in the version of D in Figure 3a. We answer it with the tree $\mathcal{T}$ in Figure 3b. The value of λ equals $\frac{4}{15}1500 = 400$, where 1500 is the ALL counter of the root.

At the beginning, $r_{total} = 0$, and the algorithm starts from the root n_1. Since $r_{total} + r_{n_1} = 600$ is greater than $\lambda = 400$, we descend to the right child n_5 of n_1, without changing r_{total}. This time, λ is greater than $r_{total}+c_{n_5}+r_{n_5} = 0+120+120 = 240$, we first increase r_{total} by $c_{n_5}+r_{n_5} = 240$, and then, visit the left child n_3 of n_5. Now, λ falls between $r_{total}+r_{n_3} = 240+120 = 360$ and $r_{total}+c_{n_3}+r_{n_3} = 240+120+120 = 480$. Hence, the algorithm terminates by returning the first value in the data domain $\mathbb{D}$ that is covered by the key k_3 of n_3. □

We now establish the correctness of our algorithm:

Lemma 4. *When Properties $P2$, $P3$ and $P4$ hold, the above algorithm correctly finds an ϵ-approximate ϕ-quantile.*

Proof. We discuss only $E > 0$ because the error-free case $E = 0$ is trivial. Let u be the value returned by our algorithm. Denote by x the number of items in the queried version $D(q)$ that are greater than or equal to u. Let $\lambda' = \phi N(q)$. To prove the lemma, we must show that x can differ from λ' by at most $\epsilon N(q)$.

Let $\mathcal{T}(q)$ be the ephemeral binary tree on $D(q)$. Recall that $\mathcal{T}(q)$ is captured by a persistent binary tree $\mathcal{T}^p$. Let v be the initial version of $\mathcal{T}^p$. Since D has incurred at most $\lfloor N(v)/2 \rfloor$ updates between versions v and q, it holds that $N(q) \geq N(v)/2$. Next, we will show $|x - \lambda'| \leq \epsilon N(v)/2$, which will imply $|x - \lambda'| \leq \epsilon N(q)$.

Consider the $\lambda = \phi \cdot ALL_{\hat{n}}$ in our query algorithm. By Property $P4$, $ALL_{\hat{n}}$ differs from $N(q)$ by at most $\epsilon N(v)/4$. This means that λ and λ' can also differ by at most $\epsilon N(v)/4$, regardless of ϕ. Hence, to prove $|x - \lambda'| \leq \epsilon N(v)/2$, it suffices to show $|x - \lambda| \leq \epsilon N(v)/4$. We distinguish three cases, depending on how u is returned (according to the pseudo-code of Figure 4) and its concrete value. In all cases, denote by P the set of nodes in $\mathcal{T}(q)$ visited by our algorithm.

Case 1: u is returned by Line 6. First, notice that if all the r- and c-counters of the nodes on P are fully accurate, then x can differ from λ by at most $\lceil B \rceil$ (Equation 3) due to Property P_2. Now, let us account for

the errors of the counters along P. The errors of each node on P can increase the difference between x and λ by at most $2E$ due to Property $P3$. Since P has at most h nodes, the total difference accumulated is at most $\lceil B \rceil + 2Eh \le \epsilon N(v)/4$.

Case 2: u is returned by Line 13, and is the smallest value in the data domain $\mathbb{D}$. So $x = N(q)$, and P involves the at most h nodes on the left-most path of $\mathcal{T}(q)$. Furthermore, λ must be greater than the final value y of r_{total} when the algorithm finishes. Note that if there is no error in the c- and r-counters of those nodes, y ought to be exactly $N(q)$. As each counter may have an error of E, after accumulating all errors, y may differ from $N(q)$ by at most $2hE \le \epsilon N(v)/8$. Therefore, λ must be at least $N(q) - \epsilon N(v)/8$. On the other hand, λ cannot exceed $ALL_{\hat{n}} \le N(q) + \epsilon N(v)/4$. So in any case $|x - \lambda| \le \epsilon N(v)/4$.

Case 3: u is returned by Line 13, but is not the smallest value in $\mathbb{D}$. Then, there is at least one node where the algorithm visited its right child. Denote by n' the *lowest* such node in P, and by S the set of nodes in P that are below n'. Also, let n be the leaf node in P. Finally, let y' be the value of r_{total} when n' was being processed at Line 5 and, as in Case 2, y be the final value of r_{total}.

We will bound the difference between x and λ by relating them to y. First, as u is returned by Line 13, λ must be greater than y. On the other hand, since the algorithm descended into the right child of n, λ must be lower than $y' + r_{n'}$, where $r_{n'}$ is the r-counter of n'. Note that $y - y'$ equals the sum of the c- and r-counters of all the nodes in S. The sum should be exactly $r_{n'}$ if all counters are accurate, and can differ from $r_{n'}$ by at most $2hE \le \epsilon N(v)/8$ if errors are accounted for. Thus, $y - y' \ge r_{n'} - \epsilon N(v)/8$. The above analysis indicates $y < \lambda < y + \epsilon N(v)/8$. Another crucial fact is that, if no counter has error, than x equals the sum of the c- and r-counters of all nodes on P, which is exactly y. Hence, the counters' errors can make x differ from y by at most $2hE \le \epsilon N(v)/8$. It therefore follows that $|x - \lambda| \le \epsilon N(v)/4$. □

3.4 Construction algorithm

We are ready to explain how to build a persistent binary tree $\mathcal{T}^p$. As before, let $D(v)$ be the first version of D covered by $\mathcal{T}^p$, i.e., $\mathcal{T}^p$ captures the ephemeral binary tree $\mathcal{T}(i)$ of $D(i)$ for each $v \le i \le v + \lfloor N(v)/2 \rfloor$.

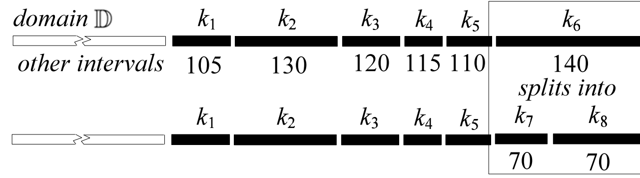
As mentioned in Section 3.1, to describe the construction of a persistence tree, it suffices to elaborate the list of updates (i.e., *insert*, *delete*, and *modify*) on an ordinary binary tree. Hence, to describe $\mathcal{T}^p$, we will explain the updates on the following binary tree $\mathcal{T}$. The initial $\mathcal{T}$ is the ephemeral tree $\mathcal{T}(v)$ on $D(v)$. For every subsequent update (i.e., an insertion or a deletion) on D, $\mathcal{T}$ is maintained accordingly, so that the current $\mathcal{T}$ always corresponds to $\mathcal{T}(v + i)$, where i is the number of updates on D that have been processed since version v. Next, we give the initialization and maintenance algorithms of $\mathcal{T}$.

Initialization. The initial $\mathcal{T} = \mathcal{T}(v)$ is built as follows. We partition the data domain $\mathbb{D}$ into several intervals, each covering $\lceil B \rceil$ (Equation 3) items of the current D, except perhaps the last interval. In this way, roughly $N(v)/B = 16/\epsilon$ intervals are obtained. $\mathcal{T}$ simply indexes all such intervals. All counters in $\mathcal{T}$ are set accurately (including the c- and r-counters of each node, as well as the ALL-counter of the root).

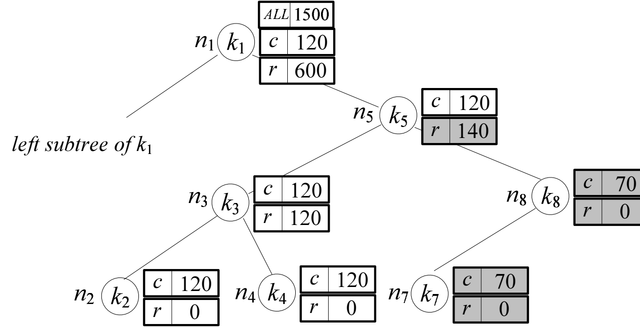
Maintenance. Given a node n in $\mathcal{T}$, denote by S_n the set of items in D that are covered by the key k_n of n. As D is updated with insertions and deletions, S_n undergoes changes, too. At all times, for each n, we maintain S_n and the accurate values c_n^*, r_n^* of counters c_n, r_n, respectively. Precisely, $c_n^* = |S_n|$ and r_n^* equals the sum of $|S_{n'}|$ of all nodes n' in the right subtree of n in $\mathcal{T}$. Note that S_n, c_n^*, and r_n^* are needed only for structure construction, and do not need to be stored in our structure.

$\mathcal{T}$ is updated only when any counter of a node n incurs error higher than E (Equation 4). There are 4 different *events*:

$E1$: $|r_n - r_n^*| > E$. In this case, we reset the r-counter of n accurately. Formally, we perform an update



(a) Split of k_6 in an Event $E3$



(b) The binary tree $\mathcal{T}$ after update

Figure 5: Updating an ephemeral binary tree

operation

$$\textit{modify}(n, t)$$

on $\mathcal{T}$, where $t = (c_n, r_n^*)$ is the new information tag of n. Note that the c-counter of n remains unchanged.

$E2$: $c_n - c_n^* > E$. Namely, c_n is greater than the accurate value by more than E. The event is dealt with by resetting *both* counters of n with an operation on $\mathcal{T}$: *modify*(n, t), where $t = (c_n^*, r_n^*)$.

$E3$: $c_n^* - c_n > E$. That is, c_n is lower than the accurate value by more than E. We create two nodes n_1 and n_2 by splitting S_n evenly (i.e., sort the items of S_n in descending order; then S_{n_1} gets the first half of the sorted list, and S_{n_2} gets the other half). Then, 3 updates are performed on $\mathcal{T}$:

$$\textit{delete}(n),\ \textit{insert}(n_1, t_1),\ \text{and}\ \textit{insert}(n_2, t_2)$$

where tags $t_1 = (c_{n_1}^*, r_{n_1}^*)$ and $t_2 = (c_{n_2}^*, r_{n_2}^*)$. Finally, the 3 operations may alter the right pointers of some nodes. For each such node n', reset its r-pointer accurately (without affecting its c-pointer) with *modify*(n', t'), where $t' = (c_{n'}, r_{n'}^*)$.

$E4$: $|ALL_n - |D|| > \epsilon N(v)/4$. In this event, n is the root $\hat{n}$ of $\mathcal{T}$. We handle the event by accurately resetting $ALL_{\hat{n}}$ and $r_{\hat{n}}$, without changing $c_{\hat{n}}$. Specifically, this is achieved with an operation *modify*$(\hat{n}, \hat{t}, |D|)$, where $\hat{t} = (c_{\hat{n}}, r_{\hat{n}}^*)$, and $|D|$ is the new value of $ALL_{\hat{n}}$.

Example 3. Next we illustrate our maintenance algorithm with an example, focusing on Event $E3$, because the handling of the other events involves only simple *modify* operations. In particular, we will continue Example 1, where Figure 3a shows the intervals (a.k.a. keys) of the nodes in $\mathcal{T}$, and Figure 3b gives the structure of $\mathcal{T}$ in detail. Also, recall that $E = 19$.

Now assume that a new value is inserted in D, and this value falls in k_6, which now covers 140 items (as in Figure 3a, k_6 covered 139 items previously). As a result, the c-counter 120 of n_6 incurs a negative error

of -20. Since the (absolute value of the) error is higher than $E = 19$, an Event $E3$ is generated. To handle the event, we first split k_6 into intervals k_7 and k_8 (see Figure 5a), each of which covers 70 items, namely, half as many as k_6. Then, node n_6 is deleted from $\mathcal{T}$, while n_7 and n_8 are inserted. The resulting structure is presented in Figure 5b. The counters of n_7 and n_8 are shaded to indicate that they are newly (accurately) set. Also shaded is the r-counter of n_5, which needs to be precisely reset because the right child of n_5 has changed (from n_6 to n_8). Note that the c-counter of n_5 is not modified. □

Now we verify that our construction algorithm guarantees Properties $P1$-$P4$.

Lemma 5. *Properties $P1$-$P4$ hold on any ephemeral binary tree $\mathcal{T}$.*

Proof. This is obvious for $P3$ and $P4$, as they are explicitly ensured by Events $E1$-$E4$. As for $P2$, it trivially holds when $\mathcal{T}$ is initiated. After that, the c-count of an existing node can be modified only in Event $E2$, which, however, cannot violate $P2$ because it decreases the count. Finally, when a new node is created by a split in Event $E3$, its c-count can be at most $(\lceil B \rceil + E + 1)/2 \le \lceil B \rceil$.

Next, we will validate $P1$ by showing that $\mathcal{T}$ can have at most $O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})$ nodes. We discuss only $E > 0$ (the case $E = 0$ is trivial). When $\mathcal{T}$ is initiated, it has $\Omega(1/\epsilon)$ nodes. Then, a new node can be created only in Event $E3$. Since D must incur at least $\lceil E \rceil$ updates to generate an $E3$, and since $\mathcal{T}$ is maintained only for $\lfloor N(v)/2 \rfloor$ updates on D, $\mathcal{T}$ can have at most $O(\frac{1}{\epsilon} + \frac{N(v)/2}{E}) = O(h/\epsilon)$ nodes. Solving $h = O(\log(h/\epsilon))$ we get $h = O(\log\frac{1}{\epsilon})$. So $\mathcal{T}$ can have at most $O(h/\epsilon) = O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})$ nodes. □

3.5 Complexity analysis

We will first show that our structure requires at most $O(\frac{M}{\epsilon H}\log^2\frac{1}{\epsilon})$ space, and then prove that our query algorithm takes $O(\log\frac{1}{\epsilon} + \log\frac{M}{H})$ time. Here, M is the total number of updates on D in history, and H (Equation 1) is the Harmonic mean of $N(1)$, $N(2)$, ..., $N(M)$, with $N(i)$ being the size of the i-th version $D(i)$ of D, $1 \le i \le M$.

Space cost. Recall that our structure consists of T persistent binary trees $\mathcal{T}_1^p$, $\mathcal{T}_2^p$, ..., $\mathcal{T}_T^p$. We will bound the space of each $\mathcal{T}_j^p$, $1 \le j \le T$, and T separately in two lemmas.

Lemma 6. *Each $\mathcal{T}_j^p$, $1 \le j \le T$, requires $O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$ space.*

Proof. As the same analysis is used for all $\mathcal{T}_j^p$, we drop the subscript j. Assume that $D(v)$ is the first version of D covered by $\mathcal{T}^p$. In other words, $\mathcal{T}^p$ supports queries on $D(v)$, $D(v+1)$, ..., $D(v + \lfloor N(v)/2 \rfloor)$. Namely, the underlying binary tree $\mathcal{T}$ of $\mathcal{T}^p$ is maintained when D evolves from $D(v)$ to $D(v + \lfloor N(v)/2 \rfloor)$. As mentioned in Section 3.1, the space of $\mathcal{T}^p$ is linear to the number of updates on $\mathcal{T}$.

Let x, y, and z be the numbers of *insert*, *delete*, and *modify* operations on $\mathcal{T}$, respectively. Next, we will show that $x + y + z = O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$. In fact, $x = O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})$ has already been established in the proof of Lemma 5. Since a deletion is always accompanied by two insertions (Event $E3$), it follows that $y = O(\frac{1}{\epsilon}\log\frac{1}{\epsilon})$. So it suffices to prove $z = O(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon})$.

A *modify* operation may be performed in all Events $E1$-$E4$. We do not need to worry, however, about those due to Event $E3$. This is because the number of right-pointer changes in a binary tree (e.g., a red-black tree) is bounded by $O(x + y)$, as argued in [8]. Each of $E1$, $E2$ and $E4$ issues $O(1)$ *modify* operations. It is easy to see that a new Event $E4$ can happen only after at least $\lceil \epsilon N(v)/4 \rceil$ updates on D since the last Event $E4$. Since totally $\lfloor N(v)/2 \rfloor$ updates occur on D during the versions covered by $\mathcal{T}^p$, Event $E4$ happens at most $O(1/\epsilon)$ times, thus generating only $O(1/\epsilon)$ *modify*.

It remains to analyze Events $E1$ and $E2$. We focus on only $E1$ because the analysis of $E2$ is similar. Event $E1$ is due to the changes of r-counters. The r-counter of a node n in $\mathcal{T}$ can have its error increased by 1, only if (i) a value in D is inserted/deleted, and (ii) the value falls in the key k_n of n (remember that k_n is an interval in the data domain). Hence, n may trigger an Event $E1$ only after $\lceil E \rceil$ updates on D since the creation of n. On the other hand, each inserted/deleted value (in D) may fall in the keys (i.e., intervals) of $O(\log \frac{1}{\epsilon})$ nodes on a single path in $\mathcal{T}$. It follows that the total number of Event $E1$'s is bounded by $O(\frac{N(v)/2}{E} \log \frac{1}{\epsilon}) = O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon})$. □

Lemma 7. $T = O(M/H)$.

Proof. Define $v(j)$, $1 \le j \le T$, be the first version $D(v)$ covered by the persistent tree $\mathcal{T}_j^p$. By our construction described at the beginning of Section 3.3, $v(1) = 1$, and for $j \ge 1$:

$$v(j+1) = \min\{M, v(j) + \lfloor N(v(j))/2 \rfloor + 1\}.$$

For notational convenience, define $v(T+1) = M + 1$.

Consider any version $i \in [v(j), v(j+1)-1]$. Since D has incurred at most $\lfloor N(v(j))/2 \rfloor$ updates between versions $v(j)$ and i, it holds that $N(i) \le 3N(v(j))/2$. With this, we have:

$$\begin{aligned}
\sum_{i=1}^{M} \frac{1}{N(i)} &= \sum_{j=1}^{T} \sum_{i=v(j)}^{v(j+1)-1} \frac{1}{N(i)} \\
&\ge \sum_{j=1}^{T} \sum_{i=v(j)}^{v(j+1)-1} \frac{2}{3N(v(j))} \\
&= \sum_{j=1}^{T} \frac{2(v(j+1) - v(j))}{3N(v(j))} \\
&\ge \sum_{j=1}^{T} \frac{N(v(j))}{3N(v(j))} = T/3
\end{aligned}$$

Hence, $M/H = \sum_{i=1}^{M} \frac{1}{N(i)} \ge T/3$, indicating $T = O(M/H)$. □

It thus follows from the above lemmas that our structure occupies $O(\frac{M}{\epsilon H} \log^2 \frac{1}{\epsilon})$ space.

Query time. The analysis of query time is trivial. Given a query, we can find the persistent tree $\mathcal{T}^p$ that covers the queried version i in $O(\log T)$ time. After that, by the same argument in [8], the ephemeral binary tree $\mathcal{T}(i)$ in $\mathcal{T}^p$ can be identified in $O(\log \frac{1}{\epsilon})$ time. Then, the query algorithm terminates after accessing a single path of $\mathcal{T}(i)$ in $O(\log \frac{1}{\epsilon})$ time. Hence, the overall query cost is $O(\log T + \log \frac{1}{\epsilon}) = O(\log \frac{1}{\epsilon} + \log \frac{M}{H})$.

Remark. The above results can be summarized as:

Theorem 3. *Given a sequence of M updates on an initially empty D, there is a structure that occupies $O(\frac{M}{\epsilon H} \log^2 \frac{1}{\epsilon})$ space, and finds any ϵ-approximate historical quantile in $O(\log \frac{1}{\epsilon} + \log \frac{M}{H})$ time.*

As a corollary, if all the updates on D are insertions (so $H = \Theta(M/\log M)$), the space and query costs become $O(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon} \log M)$ and $O(\log \frac{1}{\epsilon} + \log \log M)$, respectively. Finally, it is worth mentioning that our structure can be built efficiently in $O(M \log(\epsilon M) \log \frac{1}{\epsilon})$ time. Details will appear in the full paper.

3.6 An alternative simple solution

We close the section by pointing out an alternative structure for ϵ-approximate historical quantile search, which consumes $O(\frac{1}{\epsilon^2}\frac{M}{H})$ space. Although this is higher than the space complexity in Theorem 3, the structure has the advantage of being very simple to implement. The idea is to periodically extract $O(1/\epsilon)$ items, and use them to answer queries on $\Omega(\epsilon N(v))$ versions of D, where v is the version $D(v)$ from which the set is extracted.

Precisely, version 1 is supported by a set S_1 containing the sole item in $D(1)$. In general, if set S_i covers up to version $v-1$, we build S_{i+1} as follows:

- If $N(v) \leq 4/\epsilon$, S_{i+1} simply includes all the items in $D(v)$, and supports (queries on) only version v.
- Otherwise, S_{i+1} contains the $\lfloor k\epsilon N(v)/4 \rfloor$-th greatest items in $D(v)$ for $k = 1, 2, ..., \lceil 4/\epsilon \rceil$. S_{i+1} supports versions v, $v+1$, ..., $v + \lfloor \epsilon N(v)/4 \rfloor$.

In any case, for S_{i+1}, we also store a value ALL_{i+1}, which equals $N(v)$. Furthermore, every item in S_{i+1} keeps its rank in $D(v)$.

Given an ϵ-approximate ϕ-quantile query on version q, we identify the set S_i covering q, and return the item u in S_i whose associated rank is the closest to ϕALL_i from above. If such an item is not found (i.e., ϕALL_i exceeds the ranks of all items in S_i), the smallest value of the data domain $\mathbb{D}$ is returned. It can be verified that u is an ϵ-approximate ϕ-quantile at version q.

4 Experiments

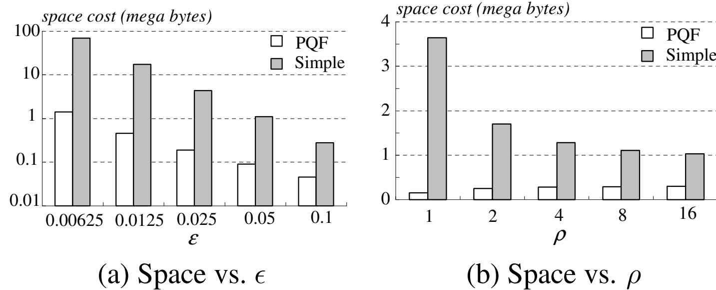
In this section, we evaluate the proposed solutions with experiments. Section 4.1 first clarifies the competing methods and how they will be compared. Then, Section 4.2 explores their characteristics using synthetic data. Finally, Section 4.3 examines their usefulness in a real networking scenario.

4.1 Competitors and metrics

In the sequel, we refer to our main method (developed in Sections 3.2-3.5) as *persistent quantile forest* (`PQF`). Since no previous solution is available for ϵ-approximate historical quantile search, we compare `PQF` with `Simple`, which is the approach presented in Section 3.6. Recall that, although `Simple` has a higher space complexity than `PQF`, it is much simpler to implement. Hence, a chief objective of the experiments is to identify when it pays off to apply `PQF`.

`PQF` and `Simple` are *synopsis structures* (just like histograms) that reside in main memory. Both of them answer a quantile query in negligible time (less than 1 μs). Therefore, we will assess (i) their space overhead, and (ii) the precision of their query results. In particular, to measure the precision of a structure, we employ a *workload* that consists of 100 queries. Each query has two parameters: the value of ϕ, and the version q queried. The ϕ of each query is randomly selected in $(0, 1]$. Furthermore, the versions of the 100 queries are placed evenly throughout the history. That is, the i-th ($1 \leq i \leq 100$) query inquires about the $\lfloor iM/100 \rfloor$-th version of the dataset, where M is the total number of updates in history. The *error* of a query is calculated as the difference between the actual rank (of the returned result) and the requested rank, in relation to the size of the queried version of the dataset. Specifically, assume that the queried version has N items, and A of them are greater than or equal to the query result; then, the error equals

$$|A - \phi N|/N.$$



(a) Space vs. ϵ (b) Space vs. ρ

Figure 6: Space comparison

This is a standard error definition in the literature (see, for example, [5, 9]). Sometimes we will report the errors of all queries directly. When this is not convenient, we report both the *average error* and *maximum error* of all the queries in a workload.

4.2 Performance characteristics

This subsection studies the general behavior of PQF and Simple. For this purpose, we generated synthetic data in a domain from 1 to $U = 2^{30}$. The update sequence of each dataset exhibits a transition of distribution involving the uniform distribution and a Gaussian distribution. Specifically, at the first version, the dataset has 100k items uniformly distributed in the domain. Then, 1 million updates are generated according to several rules. First, each update is ρ times more likely to be an insertion than a deletion, where ρ is a parameter called *insertion-deletion ratio*. As a special case, if $\rho = 1$, then roughly an equal number of insertions and deletions are created. Second, a deletion always randomly removes an existing item of the dataset. Third, if an insertion appears in the first half million updates, the inserted value follows a Gaussian distribution with mean $U/2$ and standard deviation U (a value outside the domain is discarded and re-generated). On the other hand, for an insertion that is in the second half million of updates, the inserted value is uniformly distributed in the domain. Recording all the updates of a dataset requires storage of around 8 mega bytes[3], regardless of ρ.

The first experiment compares the space consumption of PQF and Simple as a function of ϵ. We fix ρ to 1, and double ϵ from $0.1/2^4$ to 0.1. As shown in Figure 6a, PQF scales with ϵ much better than Simple. In particular, even for a tiny $\epsilon = 0.00625$, PQF requires only slightly more than 1 mega bytes, whereas Simple demands over 50 mega bytes (i.e., 6 times more than capturing all the updates precisely). This phenomenon confirms the necessity of designing a method whose space cost grows slower than $\frac{1}{\epsilon^2}$ as ϵ decreases.

Next, we study the impact of ρ on the space overhead, by setting $\epsilon = 0.1/2^2$ and doubling ρ from 1 to 16. Recall that ρ is the ratio between the numbers of insertions and deletions. The results are illustrated in Figure 6b. For a larger ρ, Simple requires less space because the Harmonic mean H (Equation 1) grows with ρ (remember that the space complexity of Simple is $O(\frac{M}{\epsilon^2 H})$). Although the space complexity of PQF is also inversely proportional to H, its space cost actually increased. This is because a higher ρ also necessitates more counter updates in PQF, which cancels the benefits of a smaller H for the range of ρ examined.

Now we proceed to evaluate the precision of each method. The next experiment employs the dataset with $\rho = 1$. Both methods are allowed storage of 80k bytes, which is around 1% of the space for recording all the updates. Figure 7 gives the errors of all queries in a workload, as a function of their versions (equivalently, the number of updates already processed when the query is issued). Notice that, for most queries, the error of PQF is lower than that of Simple by a wide margin. This is not surprising because the lower space complexity of PQF allows it to choose a smaller ϵ than Simple under the same space budget. In particular, in Figure 7, the

[3] Each update requires two integers: the inserted value and the insertion time.

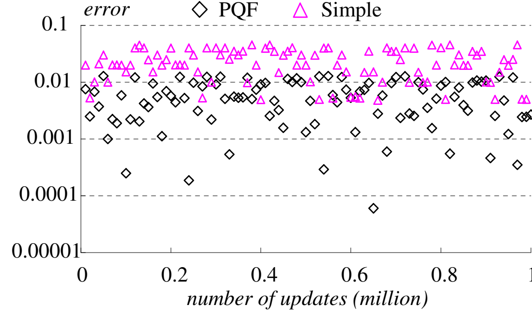
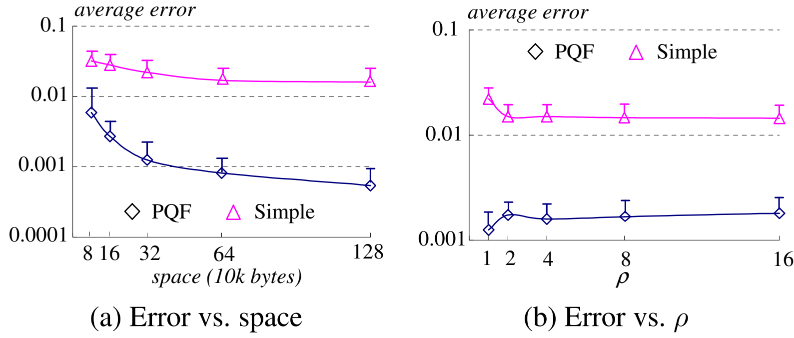


Figure 7: Error vs. the number of updates



(a) Error vs. space (b) Error vs. ρ

Figure 8: Comparison of average and maximum errors

ϵ of PQF and Simpleare 0.05 and 0.18, respectively. Observe that, for all queries, the actual errors of each method are significantly below the theoretical upper bound ϵ.

Still using the dataset with $\rho = 1$, we increase the space limit of each method, and measure their average and maximum errors (of all the queries in a workload), respectively. Figure 8a plots the average error as a function of the space budget. Each reported value has a bar on top to indicate the corresponding maximum error. The precision of PQF improves dramatically when more space is available, and is better than that of Simple by a factor more than an order of magnitude. Finally, we study the influence of the insertion-deletion ratio ρ on precision. For this purpose, we fix the space of each method to 320k bytes, and measure their average and maximum errors on the datasets with $\rho = 1, 2, ..., 16$, respectively. As shown in Figure 8b, when there are more insertions, the accuracy of PQF (Simple) is adversely (positively) affected, which is consistent with the result of Figure 6b. Nevertheless, PQF still substantially outperforms simple for all values of ρ.

4.3 Performance on real data

In this subsection, we evaluate PQF and Simple in a real network monitoring scenario. The dataset was obtained from the authors of [12], and consists of over 1.6 million packets passing through a router in the *Abilene backbone network*. For each packet, the dataset contains the timestamp the packet is received, its source and destination IPs, the ports at the source and destination, and so on. We focused on monitoring the usage distribution of the *destination ports*. Specifically, the arrival of a packet with (destination) port p increases the *frequency* of p by 1. At any snapshot, the goal is to enable quantile retrieval on the frequencies of all ports. For this purpose, each packet with port p generates a deletion followed by an insertion: we first remove the previous frequency of p, and then the add its new, incremented, frequency. The total number of updates is therefore over 3.2 million. Each port is an integer in the domain $[1, 65535]$. There are totally 53940

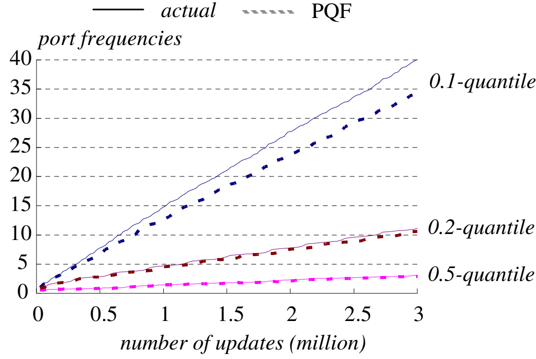


Figure 9: Exact and approximate quantiles (Abilene traffic)

distinct (destination) ports in the entire dataset. Storing all the updates demands around 25 mega bytes.

Port usage distribution is useful for detecting abnormal traffic in the network. The distribution of normal traffic should be highly skewed because some ports, such as 80 for the HTTP protocol, should be used much more frequently than others. The three solid lines in Figure 9 (in the top-down order) plot respectively the precise 0.1-, 0.2-, and 0.5-quantiles of the Abilene dataset as a function of the number of updates. The dotted lines represent the corresponding approximate quantiles retrieved from `PQF`, when it consumes 1 mega bytes of space. Both the exact and approximate quantiles clearly indicate a skewed distribution. Also observe that the *whole* frequency distribution gradually shifts upwards, which is a tough scenario for quantile computation as mentioned in [9].

Next, we compare the precision of `PQF` and `Simple` using a workload of queries (that inquire about versions scattered throughout the history). Figure 10a (10b) shows the errors of all queries when each method is allowed to use 250k (4 mega) bytes of space, which accounts for around 1% (16%) of the storage needed to capture all updates exactly. `PQF` is highly accurate in all queries, and outperforms `Simple` significantly (note that the error axis in Figure 10b is in log scale). The last experiment examines the accuracy of the two methods as the space budget doubles from 250k bytes to 4 mega bytes. Figure 11 presents each method's average and maximum errors of all the queries in a workload (in the same fashion as in Figure 8). The results confirm our earlier findings from Figure 8a.

Summarizing all the experiments, a few observations can be made regarding the performance of `PQF` and `Simple` in practice. First, for obtaining highly accurate results (with errors at the order of 0.001 or below), `PQF` must be deployed, because the space cost of `Simple` is prohibitive. Second, with space around 1% of the space of the underlying dataset, `PQF` achieves an average error around 0.01. This makes `PQF` a nice choice for query optimization in temporal databases, where an error of 0.01 is fairly acceptable. Third, if an application can accept an error between 0.05 and 0.1, `Simple` will be more appropriate due to its simplicity.

5 Conclusions and future work

This paper presented a set of formal results on the problem of ϵ-approximate historical quantile retrieval, which has not been studied previously in spite of its significant importance in practice. Our first major contribution is to establish a lower bound on the space consumption of any access method for solving the problem correctly. Besides its theoretical values, our lower bound analysis also reveals critical insights into the characteristics of the problem, including the identification of the factors affecting the space cost. As the second major contribution, we match the lower bound (up to only a square-logarithmic factor) by developing a concrete data structure to support ϵ-approximate historical quantile retrieval. Extensive experiments demonstrate

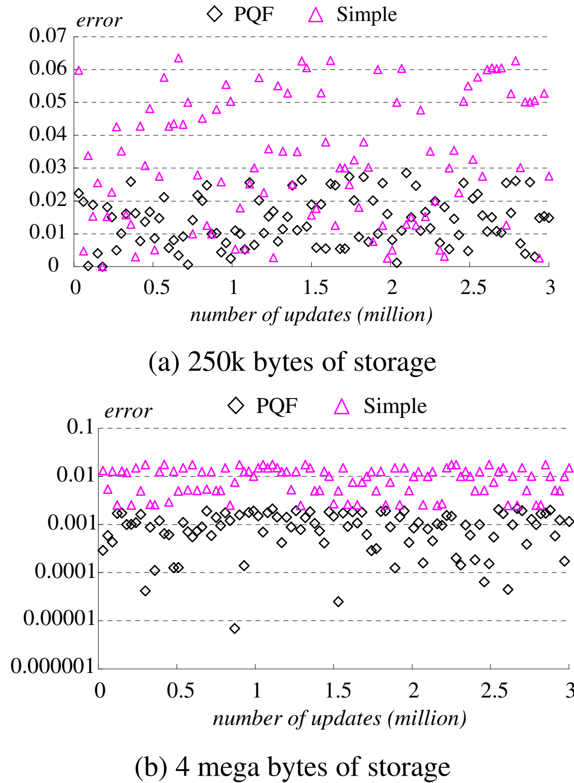


Figure 10: Error vs. the number of updates (Abilene)

that the proposed techniques are fairly effective in practice.

This work also creates several open problems that deserve further investigation. First, it remains unknown whether we can close the square-logarithmic gap between the space lower and upper bounds. A promising direction would be to apply a probabilistic approach, by allowing a method to occasionally fail in answering a query. Second, it would be natural to extend our results to other variants of quantile search (such as *biased quantiles* in [5]). Finally, in this work, we assume that the data domain can be arbitrarily large, but it is interesting to consider historical quantile retrieval on a domain with limited size. In that case, the space complexity may be substantially improved.

Acknowledgements

Yufei Tao and Cheng Sheng were supported by grants GRF4161/07, GRF 4173/08, GRF4169/09 from HKRGC, and a direct grant (2050395) from CUHK. Ke Yi was supported by a Hong Kong DAG grant (DAG07/08). Jian Pei was supported by an NSERC Discovery grant and an NSERC Discovery Accelerator Supplement grant. Feifei Li was partially supported by NSF Grant IIS-0916488.

References

[1] K. Alsabti, S. Ranka, and V. Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *VLDB*, pages 346–355, 1997.

[2] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296, 2004.

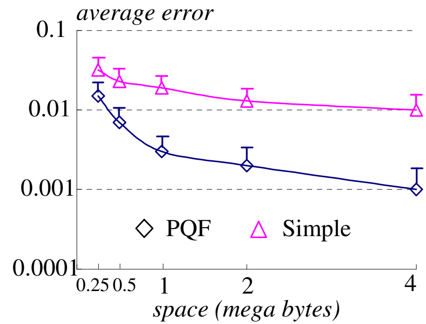


Figure 11: Average and maximum errors vs. space (Abilene)

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

[4] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *JCSS*, 7(4):448–461, 1973.

[5] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective computation of biased quantiles over data streams. In *ICDE*, pages 20–31, 2005.

[6] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[7] J. V. den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB*, pages 406–415, 1997.

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

[9] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB*, pages 454–465, 2002.

[10] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, pages 58–66, 2001.

[11] A. Gupta and F. Zane. Counting inversions in lists. In *SODA*, pages 253–254, 2003.

[12] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. D. Kolaczyk, and N. Taft. Structural analysis of network traffic flows. In *SIGMETRICS*, pages 61–72, 2004.

[13] D. B. Lomet and B. Salzberg. Access methods for multiversion data. In *SIGMOD*, pages 315–324, 1989.

[14] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *SIGMOD*, pages 426–435, 1998.

[15] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *SIGMOD*, pages 251–262, 1999.

[16] J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theo. Comp. Sci.*, 12:315–323, 1980.

[17] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Comp. Surv.*, 31(2):158–221, 1999.

[18] Y. Tao and D. Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *VLDB*, pages 431–440, 2001.

[19] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. On computing temporal aggregates with range predicates. *TODS*, 33(2), 2008.