# A Fast Algorithm for Approximate Quantiles in High Speed Data Streams

Qi Zhang and Wei Wang
University of North Carolina, Chapel Hill
Department of Computer Science
Chapel Hill, NC 27599, USA
zhangq@cs.unc.edu, weiwang@cs.unc.edu

## Abstract

*We present a fast algorithm for computing approximate quantiles in high speed data streams with deterministic error bounds. For data streams of size N where N is unknown in advance, our algorithm partitions the stream into sub-streams of exponentially increasing size as they arrive. For each sub-stream which has a fixed size, we compute and maintain a multi-level summary structure using a novel algorithm. In order to achieve high speed performance, the algorithm uses simple block-wise merge and sample operations. Overall, our algorithms for fixed-size streams and arbitrary-size streams have a computational cost of* $O(N \log(\frac{1}{\epsilon} \log \epsilon N))$ *and an average per-element update cost of* $O(\log \log N)$ *if* $\epsilon$ *is fixed.*

## 1 Introduction

Quantile computation has a wide range of applications in database and scientific computing. Computing exact quantiles on large datasets or unlimited data streams requires either huge memory or relatively slow disk-based sorting. It is proven that at least $O(N^{\frac{1}{p}})$ storage is needed for exact median (0.5 quantile) computation in $p$ passes for a data set of size $N$[12]. Recently, researchers have studied the problem of computing approximate quantiles with guaranteed error bound to improve both memory and speed performance [10, 11, 5, 6, 2, 8, 3, 4].

Streaming quantile computation has several constraints. Data streams are transient and can arrive at a high speed. Furthermore, the stream size may not be known apriori. Streaming computations therefore require single pass algorithms with small space requirement and which are able to handle arbitrary sized streams. In order to guarantee the precision of the result, the algorithm should ensure random or deterministic error bound for the quantile computation. The best reported storage bound for approximate quantile computation is $O(\frac{1}{\epsilon} \log \epsilon N)$ [5]. Many algorithms were developed for computing approximate quantiles over the entire stream history [10, 5] or over a sliding window [8, 2]; with uniform error [10, 5] or with biased error [3, 4]. However, most of these algorithms focus on reducing the space requirement and can trade off the computational cost. This can be an issue in stream applications such as streaming music, streaming video, voice over IP, etc. which require real-time performance. For high-speed streams on OC-768 links with 40 Gbps capacity, many algorithms may not have enough time for processing the data, even if the storage requirement is relieved.

In this paper, we present fast algorithms for computing approximate quantiles with uniform error on entire stream history. Specifically, for a fixed error, our algorithm has a computational cost of $O(N \log \frac{1}{\epsilon} \log \epsilon N)$ where $N$ is the size of the data stream. An average per element cost of $O(\log \log N)$ for fixed $\epsilon$ is achieved which significantly reduces the computational bandwidth requirement. Our algorithm is based on block-wise sampling and merging operations. For a fixed-sized stream with known size, we maintain a multi-level summary structure online which can answer $\epsilon$-approximate quantile query for any rank $r \in [1, n]$. For an arbitrary-sized stream with unknown size, our algorithm divides the streams into sub-streams of exponentially-increasing sizes, where a fixed-sized stream algorithm can be applied for each sub-stream. The storage requirement of our algorithms is $O(\frac{1}{\epsilon} \log^2 \epsilon N)$.

We have tested the performance of our algorithms on arbitrary-sized streams with tens of millions of el-

19th International Conference on Scientific and Statistical Database Management (SSDBM 2007)
0-7695-2868-6/07 $25.00 © 2007 IEEE

ements and different error thresholds. We also compared the performance of our algorithms against prior algorithms for arbitrary-sized streams. In practice, our algorithm is able to achieve upto 300× speedup over prior algorithms.

**Organization of the paper:** The rest of the paper is organized as follows. We describe the related work in Section 2. In Section 3, we present our algorithms and analysis for both fixed-sized streams and arbitrary-sized streams. In Section 4, we demonstrate our implementation results. Section 5 concludes the paper.

## 2 Related Work

Quantile computation has been studied extensively in the database literature. At a broad level, they can be classified as exact algorithms and approximate algorithms.

**Exact Algorithms:** Several algorithms are proposed for computing exact quantiles efficiently. There is also considerable work on deriving the lower and upper bounds of number of comparisons needed for finding exact quantiles. Mike Paterson [13] reviewed the history of the theoretical results on this aspect. The current upper bound is $2.9423N$ comparisons, and the lower bound is $(2+\alpha)N$, where $\alpha$ is the order of $2^{-40}$. Munro and Paterson [12] also showed that algorithms which compute the exact $\phi$-quantile of a sequence of $N$ data elements in $p$ passes, will need $\Omega(N^{1/p})$ space. For single pass requirement of stream applications, this requires $\Omega(N)$ space. Therefore, approximation algorithms that require sublinear-space are needed for online quantile computations on large data streams.

**Approximate Algorithms:** Approximate algorithms are either deterministic with guaranteed error or randomized with guaranteed error of certain probability. These algorithms can further be classified as uniform, biased or targetted quantile algorithms. Moreover, based on the underlying model, they can be further classified as quantile computations on entire stream history, sliding windows and distributed stream algorithms.

Jain and Chlamatac [7], Agrawal and Swami [1] have proposed algorithms to compute uniform quantiles in a single pass. However, both of these two algorithms have no *apriori* guarantees on error. Manku *et al.* [10] proposed a single pass algorithm for computing $\epsilon$-approximate uniform quantile summary. Their algorithm requires prior knowledge of $N$. The space complexity of their algorithm is $O(\frac{1}{\epsilon}\log^2 \epsilon N)$. Manku *et al.* [11] also presented a randomized uniform quantile approximation algorithm which does not require prior knowledge of $N$. The space requirement is $\frac{1}{\epsilon}(\log^2(\frac{1}{\epsilon}) + \log^2\log(\frac{1}{\delta}))$ with a failure probability of $\delta$. Greenwald *et al.* [5] improved Manku's [11] algorithm to achieve a storage bound of $O(\frac{1}{\epsilon}\log \epsilon N)$. Their algorithm can deterministically compute an $\epsilon$-approximate quantile summary without the prior knowledge of $N$. Lin *et al.* [8] presented algorithms to compute uniform quantiles over sliding windows. Arasu and Manku [2] improved the space bound using a novel exponential histogram-based data structure.

More recently, Cormode *et al.* [3] studied the problem of biased quantiles. They proposed an algorithm with poly-log space complexity based on [5]. However, it is shown in [15] that the space requirement of their algorithm can grow linearly with the input size with carefully crafted data. Cormode *et al.* [4] presented a better algorithm with an improved space bound of $O(\frac{logU}{\epsilon}\log \epsilon N)$ and amortized update time complexity of $O(\log\log U)$ where $U$ is the size of the universe where data element is chosen from and $N$ is the size of the data stream.

Recent work has also focussed on approximate quantile computation algorithms in distributed streams and sensor networks. Greenwald *et al.* [6] proposed an algorithm for computing $\epsilon$-approximate quantiles distributely for sensor network applications. Their algorithm guarantees that the summary structure at each sensor is of size $O(\log^2 n/\epsilon)$. Shrivastava *et al.* [14] presented an algorithm to compute medians and other quantiles in sensor networks using a space complexity of $O(\frac{1}{\epsilon}\log(U))$ where $U$ is the size of the universe.

To deal with massive quantile queries, Lin *et al.* [9] proposed an algorithm to reduce the number of distinct quires by clustering the queries and treating each cluster as a single query. For relative error order statistics, Zhang *et al.* [15] proposed an algorithm with confidence $1-\delta$ using $O(\frac{1}{\epsilon^2}\log\frac{1}{\delta}\log\epsilon^2 N)$ space, which improved the previous best space bound $O(\frac{1}{\epsilon^3}\log\frac{1}{\delta}\log N)$.

## 3 Algorithms

In this section, we describe our algorithms for fast computation of approximate quantiles on large high-speed data streams. We present our data structures and algorithms for both fixed-sized (with known size) and arbitrary-sized (with unknown size) streams. Furthermore, We analyze the computational complexity and the memory requirements for our algorithms.

Let $N$ denote the total number of values of the data stream and $n$ denote the number of values in the data stream seen so far. Given a user-defined error $\epsilon$ and any rank $r \in [1, n]$, an *$\epsilon$-approximate quantile* is an element in the data stream whose rank $r'$ is within

$[r - \epsilon n, r + \epsilon n]$. We maintain a summary structure to continuously answer $\epsilon$-approximate quantile queries.

### 3.1 Fixed Size Streams

We first present our approximate quantile computation algorithm for the case where $N$ is given in advance. We will generalize the algorithm with unknown $N$ in the following subsection. In practice, the former algorithm can be used for applications such as summarizing large databases that do not fit in main memory. The latter algorithm is useful for continuous streams whose size can not be predicted beforehand. We introduce our summary structure and describe the algorithm to construct the summary.

#### 3.1.1 Multi-level Quantile Summary

We maintain a multi-level $\epsilon$-summary $S$ of the stream as data elements are coming in. An *$\epsilon$-summary* is a sketch of the stream which can provide $\epsilon$-approximate answer for quantile query of any rank $r \leq n$. We maintain a multi-level summary structure $S = \{s_0, ..., s_l, ..., s_L\}$, where $s_l$ is the summary at level $l$ and $L$ is the total number of levels (see Fig.1). Basically, we divide the incoming stream into blocks of size $b$ ($b = \lfloor \frac{\log \epsilon N}{\epsilon} \rfloor$). Each level $l$ covers a disjoint bag $B_i$ of consecutive blocks in the stream, and all levels together $\bigcup B_i$ cover the whole stream. Specifically, $B_0$ always contains the most recent block (whether it is complete or not), $B_1$ contains the older two blocks, and $B_L$ consists of the oldest $2^L$ blocks. Each $s_l$ is an $\epsilon_l$-summary of $B_i$, where $\epsilon_l \leq \epsilon$.

The multi-level summary construction and maintainance is performed as follows. Initially, all levels are empty. Whenever a data element in the stream arrives, we perform the following update procedure.

1. Insert the element into $s_0$.

2. If $s_0$ is not full ($|s_0| < b$), stop and the update procedure is done for the current element. If $s_0$ becomes full ($|s_0| = b$), we reduce the size of $s_0$ by computing a sketch $s_c$ of size $\lceil \frac{b}{2} \rceil + 1$ on $s_0$. We refer to this sketch computation operation as `COMPRESS`, which we will describe in detail in later discussion. Consider $s_0$ as an $\epsilon_0$-summary of $B_0$ where $\epsilon_0 = 0$, the `COMPRESS` operation guarantees that $s_c$ is an $(\epsilon_0 + \frac{1}{b})$-summary. After `COMPRESS` operation, $s_c$ is sent to level 1.

3. If $s_1$ is empty, we set $s_1$ to be $s_c$ and the update procedure is done. Otherwise, we merge $s_1$ with $s_c$ which is sent by level 0 and empty $s_0$. We refer to these operations as `MERGE` on $s_1, s_c$ and `EMPTY` on $s_0$. Generally, the `MERGE(`$s_{l+1}$`, `$s_c$`)` operation merges $s_{l+1}$ with the sketch $s_c$ by performing a merge sort. The `EMPTY(`$s_l$`)` operation empties $s_l$ after `MERGE` operation is finished. Finally, we perform `COMPRESS` on the result of `MERGE`, and send the resulting sketch $s_c$ to level 2.

4. If $s_2$ is empty, we set $s_2$ to be $s_c$ and the update procedure is done. Otherwise, we perform the operations $s_2$ =`MERGE(`$s_2$, $s_c$`)`, $s_c$ =`COMPRESS(`$s_2$`)`, `EMPTY(`$s_1$`)` in the given order, and send new $s_c$ to level 3.

5. We repeatedly perform step 4 for $s_i, i = 3, \ldots, L$ until we find a level $L$ where $s_L$ is empty.

The pseudo code of the entire update procedure whenever an element $e$ comes is shown in Algorithm 1. In the following discussion, we describe the operations `COMPRESS, MERGE` in detail.

Assume that $s$ is an $\epsilon$-summary of stream $B$. For each element $e$ in $s$, we maintain $rmax(e)$ and $rmin(e)$ which represent the maximum and minimum possible ranks of $e$ in $B$, respectively. Therefore, we can answer the $\epsilon$-approximate quantile query of any rank $r$ by returning the value $e$ which satisfies: $rmax(e) \leq r + \epsilon|B|$ and $rmin(e) \geq r - \epsilon|B|$. Initially, $rmin(e) = rmax(e) = rank(e)$. $rmin(e), rmax(e)$ are updated during the `COMPRESS` and `MERGE` operations.

`COMPRESS`$(s, \frac{1}{b})$: The `COMPRESS` operation takes at most $\lceil \frac{b}{2} \rceil + 1$ values from $s$, which are: $quantile(s, 1)$, $quantile(s, \lfloor \frac{2|B|}{b} \rfloor)$, $quantile(s, \lfloor 2\frac{2|B|}{b} \rfloor)$, ...,$quantile(s, \lfloor i\frac{2|B|}{b} \rfloor)$,..., $quantile(s, |B|)$, where $quantile(s, r)$ queries summary $s$ for quantile of rank $r$. According to [6], the result of `COMPRESS`$(s, \frac{1}{b})$ is an $(\epsilon + \frac{1}{b})$-summary, assuming $s$ is an $\epsilon$-summary.

`MERGE`$(s, s')$: The `MERGE` operation combines $s$ and $s'$ by performing a merge-sort on $s$ and $s'$. According to [6], if $s$ is an $\epsilon$-summary of $B$ and $s'$ is an $\epsilon'$-summary of $B'$, the result of `MERGE`$(s, s')$ is an $\overline{\epsilon}$-summary of $B \bigcup B'$ where $\overline{\epsilon} = max(\epsilon, \epsilon')$.

**Lemma 1.** *The number of levels in the summary structure is less than* $\log(\epsilon N)$.

*Proof.* In the entire summary structure construction, $s_0$ becomes full at most $\frac{N}{b}$ times, $s_l$ becomes full $\frac{N}{2^l b}$ times and the highest level $s_L$ becomes full at most once. Therefore,

$$L \leq \log(\frac{N}{b}) < \log(\epsilon N) - \log(\log(\epsilon N)) < \log(\epsilon N) \quad (1)$$

. □

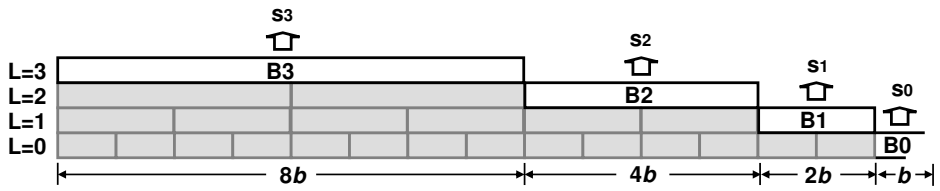



**Figure 1. Multi-level summary $S$: This figure highlights the multi-level structure of our $\epsilon$-summary $S = \{s_0, s_1, \ldots, s_L\}$. The incoming data is divided into equi-sized blocks of size $b$ and blocks are grouped into disjoint bags, $B_0, B_1, \ldots, B_l, \ldots, B_L$ with $B_l$ for level $l$. $B_0$ contains the most recent block, $B_1$ contains the older two blocks, and $B_L$ consists of the oldest $2^L$ blocks. At each level, we maintain $s_l$ as the $\epsilon_l$-summary for $B_l$. The total number of levels $L$ is no more than $\log \frac{N}{b}$**

.

**Algorithm 1** Update($e$,$S$,$\epsilon$)
**Input** $e$: current data element to be inserted, $S$: current summary structure $S = \{s_0, \ldots, s_l, \ldots, s_L\}$, $\epsilon$: required approximation factor of $S$

```
1: insert e into s_0
2: if |s_0| = b then
3:    sort s_0
4:    s_c ← compress(s_0, 1/b)
5:    empty(s_0)
6: else
7:    exit
8: end if
9: for l = 1 to L do
10:   if |s_l| = 0 then
11:      s_l ← s_c
12:      break
13:   else
14:      s_c = compress(merge(s_l, s_c), 1/b)
15:      empty(s_l)
16:   end if
17: end for
```

**Lemma 2.** *Each level in our summary maintains an error less than $\epsilon$.*

*Proof.* During the construction process of $S$, the error at each level $\epsilon_l$ depends on the `COMPRESS` and `MERGE` operations. Intially, $\epsilon_0 = 0$. At each level, `COMPRESS`($s_l$, $\frac{1}{b}$) operation generates a new sketch $s_c$ with error $\epsilon_l + \frac{1}{b}$ and added to level $l+1$, and `MERGE` does not increase the error. Therefore, the error of the summary in $s_{l+1}$ is given by

$$\epsilon_{l+1} = \epsilon_l + \frac{1}{b} = \epsilon_0 + \frac{l+1}{b} = \frac{l+1}{b} \tag{2}$$

From equations 2 and 1, it is easy to verify that

$$\epsilon_l = \frac{l}{b} < \frac{\log(\epsilon N)}{\frac{\log(\epsilon N)}{\epsilon}} = \epsilon \tag{3}$$

□

To answer a query of any rank $r$ using $S$, we first sort $s_0$ and merge the summaries at all levels $\{s_l\}$ using the `MERGE` operation, denote it as `MERGE`($S$). Then the $\epsilon$-approximate quantile for any rank $r$ is the element $e$ in `MERGE`($S$) which satisfies: $rmin(e) \geq r - \epsilon N$ and $rmax(e) \leq r + \epsilon N$.

**Theorem 1.** *For multi-level summary $S$, `MERGE`($S$) is an $\epsilon$-approximate summary of the entire stream.*

*Proof.* `MERGE` operation on all $s_l$ generates a summary for $\bigcup B_l$ with approximation factor $\epsilon_U = \max(\epsilon_1, \epsilon_2, \ldots, \epsilon_L)$. According to Lemma 2, $\epsilon_U < \epsilon$. Since the union of all the $B_l$ is the entire stream, `MERGE`($S$) is an $\epsilon$-approximate summary of the entire stream. □

### 3.1.2 Performance Analysis

Our summary structure maintains at most $b+3$ elements in each level (after `MERGE` operation) and there are $L$ levels in the summary structure. Therefore, the storage requirement for constructing the summary is bounded by $(b+3)L = O(\frac{1}{\epsilon}\log^2(\epsilon N))$. The storage requirement for our algorithm is higher than the best storage bound proposed by Greenwald and Khanna [5], which is $O(\frac{1}{\epsilon}\log(\epsilon N))$. However, The goal behind our algorithm is to achieve faster computational time with reasonable storage. In practice, the memory size requirements for the algorithm can be a small fraction of the RAM on most PCs even for peta-byte-sized datasets (see table 1).

**Theorem 2.** *The average update cost of our algorithm is $O(\log(\frac{1}{\epsilon}\log(\epsilon N)))$.*

*Proof.* At level 0, for each block we perform a sort and a `COMPRESS` operation. The cost of sort per block is $b \log b$, `COMPRESS` per block is $\frac{b}{2}$. Totally, there are $\frac{N}{b}$

blocks, so the total cost at level 0 is: $N \log b + \frac{N}{2}$. At each level $L_i, i > 0$, we perform a `COMPRESS` and a `MERGE` operation. Each `COMPRESS` costs $b$, since a linear scan is required to batch query all the values needed (refer to `COMPRESS` operation). Each `MERGE` costs $b$ with a merge sort. In fact, the computation cost of `MERGE` also includes the updates of $rmin$ and $rmax$ (will be discussed in Sec 3.3), which can be done in linear time. Thus the cost of a `MERGE` adds up to $2b$. Therefore, the total expected cost of computing the summary structure is $N \log b + \frac{N}{2} + \sum_{i=1}^{i=L} \frac{N}{2^i b} 3b = O(N \log(\frac{1}{\epsilon} \log(\epsilon N))$. The average update time per element is $O(\log(\frac{1}{\epsilon} \log(\epsilon N)))$.

□

In practice, for a fixed $\epsilon$, the average per element computation cost of our algorithm is given by $O(\log \log N)$ and the overall computation is almost linear in performance. The algorithm proposed by Greenwald and Khanna [5] has a best case computation time (per element) of $O(\log s)$, and worst computation time (per element) of $O(s)$ where $s$ is $\frac{1}{\epsilon} \log(\epsilon N)$. We will demonstrate in our experiment section the comparison of the performance.

The majority of the computation in the summary construction is dominated by the sort operations on blocks. Although sorting is computationally intensive, it is fast on small blocks which fit in the CPU L2 caches. Table 1 shows a comparison of the block size, memory requirement as a function of stream size $N$ with error bound 0.001 using our generalized streaming algorithm in the next section. In practice, the size of the blocks in our algorithm is smaller than the CPU cache size even for peta-byte-sized data streams.

## 3.2 Generalized Streaming Algorithm

We generalize our algorithm for fixed size streams to compute approximate quantiles in streams without prior knowledge of size $N$. The basic idea of our algorithm is as follows. We partition the input stream $P$ into disjoint sub-streams $P_0, P_1, \ldots, P_m$ with increasing size. Specifically, sub-stream $P_i$ has size $\frac{2^i}{\epsilon}$ and covers the elements whose location is in the interval $[\frac{2^i-1}{\epsilon}, \frac{2^{i+1}-1}{\epsilon})$. By partitioning the incoming stream into sub-streams with known size, we are able to construct a multi-level summary $S_i$ on each sub-stream $P_i$ using our algorithm for fixed size streams. Our summary construction algorithm is as follows.

1. For the latest sub-stream $P_k$ which has not completed, we maintain a multi-level $\epsilon'$-summary $S_C$ using Algorithm 1 by performing $Update(e, S_C, \epsilon')$ whenever an element comes. Here $\epsilon' = \frac{\epsilon}{2}$.

**Algorithm 2** gUpdate($e$,$\overline{S}, \epsilon, S_C$)
**Input** $e$: current data element, $\overline{S}$: current summary structure, $\overline{S} = \{\overline{S}_0, \overline{S}_1, \ldots, \overline{S}_{k-1}\}$ (sub-streams $P_0, \ldots, P_{k-1}$ have completely arrived), $\epsilon$: required approximation factor of $\overline{S}$, $S_C$: the fixed size multi-level summary corresponding to the current sub-stream $P_k$, $S_C = \{s_0, s_1, \ldots, s_L\}$

```
1: if e is the last element of P_k then
2:    Apply merge on all the levels of S_C: s_all = merge(S_C) = merge(s_0, s_1, ..., s_L)
3:    S̄_k = compress(s_all, ε/2)
4:    S̄ = S̄ ⋃ {S̄_k}
5:    S_C ← φ
6: else
7:    update S_C: S_C = Update(e, S_C, ε/2)
8: end if
```

2. Once the last element of sub-stream $P_k$ arrives, we compute an $\frac{\epsilon}{2}$-summary on `MERGE`$(S_C)$, which is the merged set of all levels in $S_C$. The resulting summary $\overline{S}_k$ =`COMPRESS`(`MERGE`$(S_C)$, $\frac{\epsilon}{2}$) is an $\epsilon$-summary of $P_k$ and it consists of $\frac{2}{\epsilon}$ elements.

3. The ordered set of the summaries of all complete sub-streams so far $\overline{S} = \{\overline{S}_0, \overline{S}_1, \ldots, \overline{S}_{k-1}\}$ is the current multi-level $\epsilon$-summary of the entire stream except the incomplete sub-stream $P_k$.

The pseudo code for the update algorithm for stream with unknown size is shown in Algorithm 2. Initially, $\overline{S} = \phi$. Whenever an element comes, *gUpdate* is performed to update the summary structure $\overline{S}$.

To answer a query of any rank $r$ using $\overline{S}$, if $S_C$ is not empty, we first compute $\overline{S}_k$ for the incomplete sub-stream $P_k$: $\overline{S}_k = compress(merge(S_C), \frac{\epsilon}{2})$, then we merge all the $\epsilon$-summaries $\overline{S}_0, \overline{S}_1, \ldots, \overline{S}_{k-1}$ in $\overline{S}$ together with $\overline{S}_k$ using `MERGE` operation, the final summary is the $\epsilon$-summary for $P$.

### 3.2.1 Performance Analysis

We first present the storage analysis and then analyze the computational complexity of our algorithm.

**Theorem 3.** *The space requirement of Algorithm 2 is* $O(\frac{1}{\epsilon} \log^2(\epsilon n))$.

*Proof.* At any point of time, assume that the number of data elements arriving so far is $n$. According to Algorithm 1, we compute and maintain a multi-level $\epsilon$-approximate summary $S_C$ for the current sub-stream $P_k$. For each of the previous sub-streams $P_i, i = 1, \ldots, k-1$ which are complete, we maintain an $\epsilon$-summary $\overline{S}_i$ of size $\frac{2}{\epsilon}$. Since $k \leq \lfloor \log(\epsilon n +$

| Stream size (N) | Maximum Block Size (Bytes) | Bound of Number of Tuples | Bound of Summary Size (Bytes) |
|---|---|---|---|
| $10^6$ | 191.2KB | 161K | 1.9MB |
| $10^9$ | 420.4KB | 717K | 8.6MB |
| $10^{12}$ | 669.6KB | 1.67M | 20MB |
| $10^{15}$ | 908.8KB | 3.03M | 36.4MB |

**Table 1. This table shows the memory size requirements of our Generalized algorithm (with unknown size) for large data streams with an error of 0.001. Each tuple consists of a data value, and its minimum and maximum rank in the stream, totally 12 bytes. Observe that the block size is less than a MB and fits in the L2 cache of most CPUs. Therefore, the sorting will be in-memory and can be conducted very fast. Also, the maximum memory requirement for our algorithm is a few MB even for handling streams of 1 peta data.**

$1)\rfloor$, totally we need $O(\frac{1}{\epsilon}\log \epsilon n)$ space. According to the space bound for fixed size streams, we need $O(\frac{1}{\epsilon}\log^2(\epsilon n))$ space for computing the summary $S_C$ for the current sub-stream. Therefore, the space requirement for the entire algorithm at any point of time is $O(\frac{1}{\epsilon}\log^2(\epsilon n))$. □

**Theorem 4.** *The average update cost of Algorithm 2 is* $O(\log(\frac{1}{\epsilon}\log \epsilon n))$.

*Proof.* According to Theorem 2, the computational complexity of each sub-stream $P_i, i = 0, 1, \ldots, \lfloor \log(\epsilon n + 1)\rfloor$ is $O(n_i \log(\frac{1}{\epsilon'}\log(\epsilon' n_i)))$ where $n_i = |P_i| = \frac{2^i}{\epsilon}, \Sigma n_i = n, \epsilon' = \frac{\epsilon}{2}$. After each sub-stream $P_i$ is complete, we perform an additional `MERGE` and `COMPRESS` operation each of cost $O(\frac{1}{\epsilon'}\log^2(\epsilon' n_i))$ to construct $\overline{S}_i$.

Given the above observations, the total computational cost of our algorithm is

$$\sum_{i=0}^{i=\lfloor \log(\epsilon n+1)\rfloor} (\frac{2^i}{\epsilon}\log(\frac{2(i-1)}{\epsilon}) + \frac{2}{\epsilon}(i-1)^2) \quad (4)$$

.

Simplifying equation 4, the total computational cost of our algorithm is $O(n\log(\frac{1}{\epsilon}\log(\epsilon n)))$, the average updating time per element is $O(\log(\frac{1}{\epsilon}\log(\epsilon n)))$, which is $O(\log\log n)$ if $\epsilon$ is fixed. □

### 3.3 Update $rmin(e)$ and $rmax(e)$

For both fixed size stream and arbitrary size stream, to answer the quantile query, we need to update $rmin$ and $rmax$ values of each element $e$ in the summary properly.

$rmin(e)$ and $rmax(e)$ are updated during `COMPRESS` and `MERGE` operations as follows (as in [6]):

**Rank update in** `MERGE`: Let $S' = x_1, x_2, \ldots, x_a$ and $S'' = y_1, y_2, \ldots, y_b$ be two quantile summaries. Let $S = z_1, z_2, \ldots, z_{a+b} = \texttt{MERGE}(S', S'')$. Assume $z_i$ corresponds to some element $x_r$ in $Q'$. Let $y_s$ be the largest element in $S''$ that is smaller than $x_r$ ($y_s$ is undefined if no such element), and let $y_t$ be the smallest element in $S''$ that is larger than $x_r$ ($y_t$ is undefined if no such element). Then

$$rmin_S(z_i) = \begin{cases} rmin_{S'}(x_r) & \text{if } y_s \text{ undefined} \\ rmin_{S'}(x_r) + rmin_{S''(y_s)} & \text{otherwise} \end{cases}$$

$$rmin_S(z_i) = \begin{cases} rmax_{S\prime}(x_r) + rmax_{S\prime\prime(y_s)} & \text{if } y_t \text{ undefined} \\ rmax_{S\prime}(x_r) + rmax_{S\prime\prime(y_t)} - 1 & \text{otherwise} \end{cases}$$

**Rank update in** `COMPRESS`: Assume $\texttt{COMPRESS}(S') = S$, for any element $e \in S$, we define $rmin_S(e) = rmin_{S'}(e)$ and $rmax_S(e) = rmax_{S'}(e)$.

## 4 Implementation and Resuts

We implemented our algorithms in C++ on an Intel 1.8 GHz Pentium PC with 2GB main memory. We tested our algorithm with a C++ implementation of the algorithm in [5] (refer to as GK01 in the remaining part of the paper) from the authors.

### 4.1 Results

We measured the performance of GK01 and our algorithm on different data sets. Specifically, we studied the computational performance as a function of the size of the incoming stream, the error and input data distribution. In all experiments, we do not assume the knowledge of the stream size, and we use *float* as data type which takes 4 bytes.

#### 4.1.1 Sorted Input

We tested our algorithms using an input stream with either sorted or reverse sorted data. Fig. 2(a) shows the performance of GK01 and our algorithm as the input data stream size varies from $10^6$ to $10^7$ with a guaranteed error bound of 0.001. For these experiments,

as the data stream size increases, the block size in the largest sub-stream varies from 191.2K to 270.9K. In practice, our algorithm is able to compute the summary on a stream of size $10^7(40MB)$ using less than 2MB RAM. Our algorithm is able to achieve a $200-300\times$ speedup over GK01. Note that the sorted and reverse sorted curves for GK01 are almost overlapping due to the log-scale presentation and small difference between them (average 1.16% difference). Same reason for the sorted and reverse sorted curves for our algorithm, and the average difference between them is 2.1%.

We also measured the performance of our algorithm and GK01 by varying the error bound from $10^{-3}$ to $10^{-2}$ on sorted and reverse sorted streams. Fig. 2(b) shows the performance of our algorithm and GK01 on an input stream of $10^7$ data elements. We observe that the performance of our algorithm is almost constant even when the approximation accuracy of quantiles increases by $10\times$. Note that the performance of GK01 is around $60\times$ slower for large error and around $300\times$ slower for higher precision quantiles compared with our algorithm.

#### 4.1.2 Random Input

In order to measure the average case performance, we measured the performance of our algorithm and GK01 on random data. Fig. 3(a) shows the performance of GK01 and our algorithm as the input data stream size varies from $10^6$ to $10^7$ with error bound of 0.001. As the data size increases, the time taken by our algorithm increases almost linearly as the computational requirement of our algorithm is $\mathrm{O}(n \log\log n)$. We observe that our algorithm is able to achieve about $200-300\times$ speedup over GK01.

In Fig. 3(b), we evaluated our algorithms on a data stream size of $10^7$ by varying the error bound from $10^{-2}$ to $10^{-3}$. We observe that the performance of our algorithm degrades by less than 10% while computing a summary with $10\times$ higher accuracy. This graph indicates that the performance of our algorithm is sublinear to the inverse of the error bound. In comparison, the performance of GK01 algorithm degrades by over 500% as the accuracy of the computed summary increases by $10\times$. In practice, the computational time increase for computing a higher accuracy summary using our algorithm is significantly lower than that using GK01.

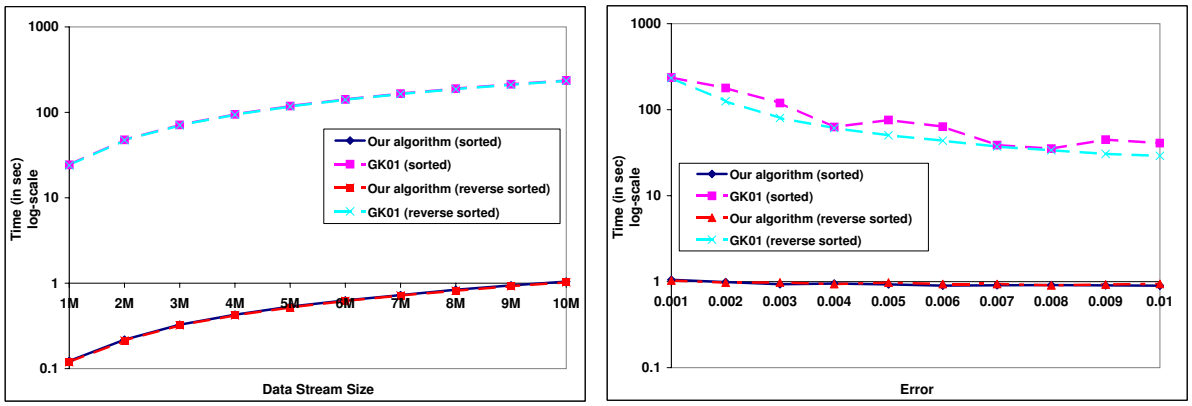
### 4.2 Analysis

The worst-case storage requirement for our algorithm is $O(\frac{1}{\epsilon}\log^2(\epsilon N))$. It is comparable to the storage requirement of MRL [10] and higher than GK01. Although the storage requirement is comparatively high, for many practical applications, the storage used by our algorithm is small enough to manage. For example, a stream with 100 million values and error bound 0.001 has a worst-case storage requirement of 5MB and practical on most PCs. Although our algorithm has a higher storage requirement than GK01, our algorithm can construct the summary upto two orders of magnitude faster than GK01. In terms of the computational cost, our algorithm has an expected cost of $\mathrm{O}(n\log(\frac{1}{\epsilon}\log(\epsilon N)))$. Therefore, for a fixed error bound, the algorithm has an almost linear increase in computational time in $n$. Our algoritm also has a near-logarithmic increase in time as error bound decreases. Therefore, our algorithm is able to handle higher accuracy, large data streams efficiently.

## 5 Conclusion and Future Work

We presented fast algorithms for computing approximate quantiles for streams. Our algorithms are based on simple block-wise merge and sort operations which significantly reduces the update cost performed for each incoming element in stream. In order to handle unknown size of the stream, we divide the incoming streams into sub-streams of exponentially increasing sizes. We construct summaries efficiently using limited space on the sub-streams. For both fixed sized and arbitrary sized streams, our algorithm has an average update time complexity of $O(\log\frac{1}{\epsilon}\log\epsilon N)$. We also analyzed the performance of prior algorithms. We evaluated our algorithms on different data sizes and compared them with optimal implementations of prior algorithms. In practice, our algorithm can achieve up to $300\times$ improvement in performance. Moreover, our algorithm exhibits almost linear performance with respect to stream size and performs well on large data streams.

There are many interesting problems for future investigation. We would like to extend our block-wise merge and compress scheme to compute quantiles quickly over sliding windows. Another interesting problem is to extend our current algorithm to handle biased quantiles. We are also interested in designing biased quantile algorithms on distributed streams and sensor networks. We would also like to design better streaming algorithms with higher computational performance for other problems such as k-median clustering, histograms, etc.




(a) *Summary construction time vs stream size*

(b) *Summary construction time vs error*

**Figure 2.** ***Sorted Data:*** **We used the sorted and reverse sorted input data to measure the best possible performance of the summary construction time using our algorithm and GK01. Fig. 2(a) shows the computational time as a function of the stream size on a log-scale for a fixed epsilon of** $0.001$**. We observe that the sorted and reverse sorted computation time curves for GK01 are almost overlapping due to the log-scale presentation and small difference between them (average** $1.16\%$ **difference). Same reason for the sorted and reverse sorted curves for our algorithm, and the average difference between them is** $2.1\%$**. We also observe that the performance of our algorithm is almost linear and the computational performance is almost two orders of magnitude faster than GK01. Fig. 2(b) shows the computational time as a function of the error. We observe the higher performance of our algorithm which is** $60-300\times$ **faster than GK01. Moreover, GK01 has a significant performance overhead as the error becomes smaller.**

## Acknowledgements

We would like to thank Dr. Michael B. Greenwald for providing the optimized GK01 code and many useful suggestions.

## References

[1] R. Agrawal and A. Swami. A one-pass space-efficient algorithm for finding quantiles. *International Conference of Management of Data(COMAD)*, 1995.

[2] A. Arasu and G. S. Manku. Approximate counts and quantiles over sliding windows. *ACM Symposium on Principles of Database Systems (PODS)*, pages 286–296, 2004.

[3] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Effective computation of biased quantiles over data streams. *International Conference on Data Engineering*, pages 20–32, 2005.

[4] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava. Space- and time-efficient deterministic algorithms for biased quantiles over data streams. *The ACM Symposium on Principles of Database Systems PODS*, pages 263–272, 2006.

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

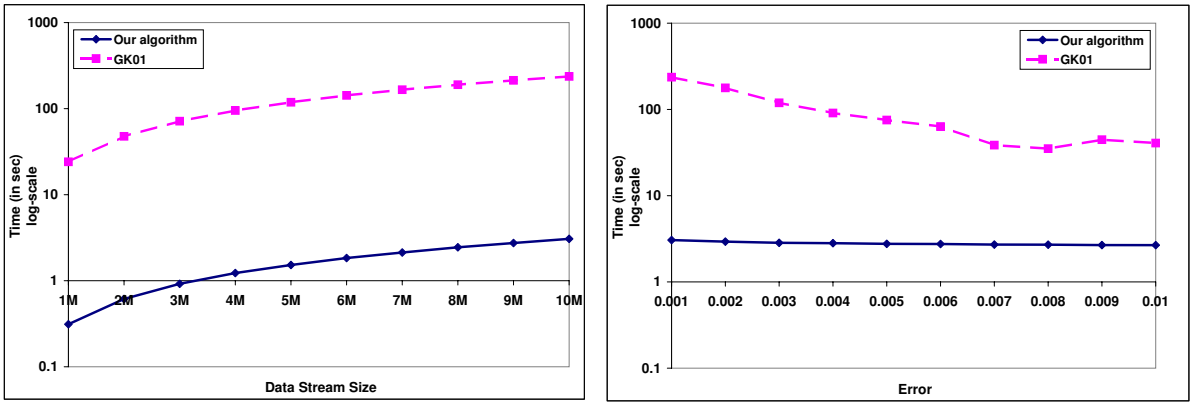
[6] M. B. Greenwald and S. Khanna. Power-conserving computation of order-statistics over sensor networks. *Symposium on Principles of database systems*, 2004.

[7] R. Jain and I. Chlamtac. The $p^2$ algorithm for dynamic calculation for quantiles and histograms without storing observations. *Communications of the ACM*, 28(10):1076–1085, 1985.

[8] X. Lin, H. Lu, J. Xu, and J. X. Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, page 362, Washington, DC, USA, 2004. IEEE Computer Society.

[9] X. Lin, J. Xu, Q. Zhang, H. Lu, J. X. Yu, X. Zhou, and Y. Yuan. Approximate processing of massive continuous quantile queries over high-speed data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(5):683–698, 2006.

[10] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD*, 12:426–435, June 1998.




(a) *Summary construction time vs stream size*

(b) *Summary construction time vs error*

**Figure 3.** ***Random Data:*** **We used the random input data to measure the performance of the summary construction time using our algorithm and GK01. Fig. 3(a) shows the computational time as a function of the stream size on a log-scale for a fixed epsilon of** $0.001$**. We observe that the performance of our algorithm is almost linear. Furthermore, the log-scale plot indicates that our algorithm is almost two orders of magnitude faster than GK01. Fig. 3(b) shows the computational time as a function of the error. We observe that our algorithm is almost constant whereas GK01 has a significant performance overhead as the error becomes smaller.**


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

[12] J. I. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[13] M. Paterson. Progress in selection. *Scandinavian Workshop on Algorithm Theory*, 1997.

[14] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249, New York, NY, USA, 2004. ACM Press.

[15] Y. Zhang, X. Lin, J. Xu, F. Korn, and W. Wang. Space-efficient relative error order sketch over data streams. *International Conference on Data Engineering*, 2006.