# Quancurrent: A Concurrent Quantiles Sketch

Shaked Elias Zada
Technion
Haifa, Israel
shakeli@alumni.technion.ac.il

Arik Rinberg*
Technion
Haifa, Israel
arikrinberg@google.com

Idit Keidar
Technion
Haifa, Israel
idish@ee.technion.ac.il

## ABSTRACT

Sketches are a family of streaming algorithms widely used in the world of big data to perform fast, real-time analytics. A popular sketch type is Quantiles, which estimates the data distribution of a large input stream. We present Quancurrent, a highly scalable concurrent Quantiles sketch. Quancurrent's throughput increases linearly with the number of available threads, and with 32 threads, it reaches an update speedup of 12x and a query speedup of 30x over a sequential sketch. Quancurrent allows queries to occur concurrently with updates and achieves an order of magnitude better query freshness than existing scalable solutions.

## CCS CONCEPTS

• **Theory of computation → Concurrent algorithms**; *Sketching and sampling*; Streaming, sublinear and near linear time algorithms.

## KEYWORDS

big data; streaming algorithms; sketches; quantiles; real-time analysis; concurrency

**ACM Reference Format:**
Shaked Elias Zada, Arik Rinberg, and Idit Keidar. 2023. Quancurrent: A Concurrent Quantiles Sketch. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '23), June 17–19, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3558481.3591074

## 1 INTRODUCTION

Data sketches, or *sketches* for short, are indispensable tools for performing analytics on high-rate, high-volume incoming data streams [10]. Sketches are designed for stream settings in which each data item is only processed once. A sketch data structure is essentially a succinct (sublinear) summary of a data stream that approximates a specific query (unique element count, quantile values, etc.).

With the rise of big data, a fundamental task in data management and analysis is to describe the distribution of the data. This is used in applications such as exploratory data analysis [22], operation monitoring [4], and more. Quantile approximation is a nonparametric representation, widely used to characterize data distributions [23, 26].

The Quantiles sketch family captures this task [5, 9, 12, 17]: In a stream of $n$ elements, for any $0 \leq \phi \leq 1$, a query for quantile $\phi$ returns an estimate of the $\lfloor n\phi \rfloor^{\text{th}}$ largest element. For example, quantile $\phi = 0.5$ is the median. Due to the importance of quantiles approximation, Quantiles sketches are a part of many analytics platforms, e.g., Druid [11], Hillview [8], Presto [18], and Spark [20].

Our goal in this paper is to build a scalable multi-threaded Quantiles sketch. Specifically, we parallelize the mergeable Quantiles sketch proposed by Agarwal et al. [5], which is popular e.g., implemented in the Apache DataSketches open-source library [1], relatively simple, and forms a basis for follow-up works [16, 23]. This sketch is of sublinear-size and provides *probably approximately correct (PAC)* estimates, which approximate a quantile within some error $\epsilon n$ with a failure probability bounded by some parameter $\delta$. Section 2, provides background on the sequential sketch, and Section 3 then presents Quancurrent, our highly scalable concurrent Quantiles sketch.

The vast majority of the literature on sketches considers a sketch built by a single thread, where queries are served after the sketch construction is complete. Only recently, we begin to see works leveraging parallel architectures to achieve a higher ingestion throughput while also enabling queries concurrently with updates [19, 21]. Of these, the only solution applicable to quantiles that we are aware of is the Fast Concurrent Data Sketches (FCDS) framework proposed by Rinberg et al. [19]. FCDS is based on local buffering of updates by multiple worker threads and a single *propagator* thread constantly propagates elements from all local buffers to a shared global sketch. Queries access the global sketch and return approximations based on a subset of the stream processed so far, including all elements that has been propagated into the global sketch. The *freshness* the query is governed by the size of the local buffers.

The FCDS paper [19] provides a generic parallel sketch construction so is applicable also for quantiles. Nevertheless, when FCDS is used for quantiles, the process of propagation includes a heavy merge-sort, therefore, by using a single propagator, a sequential bottleneck is formed. Consequently, large local buffers are required to offset the heavy sorting and keep the working threads busy during propagations resulting low query freshness. The scalability of FCDS-based Quantiles sketches is thus limited unless large buffers are used causing query freshness to be heavily compromised (as we show in Section 5). Note, no FCDS-based Quantiles Sketch implementation was evaluated in the sketches paper or included as part of the FCDS contribution to the Apache DataSketches open-source library [1]. Our goal in this work is to provide a scalable concurrent Quantiles sketch that retains a small error bound with reasonable query freshness. We are currently in the process of contributing

*This work was completed as a PhD candidate at the Technion.

our concurrent Quantiles sketch to the Apache DataSketches open-source library [1].

Like FCDS, Quancurrent relies on local buffering of stream elements, which are then propagated in bulk to a shared sketch. But Quancurrent improves on FCDS by eliminating the latter's sequential propagation bottleneck, which mostly stems from the need to sort large buffers.

In Quancurrent, sorting occurs at three levels – a small thread-local buffer, an intermediate NUMA-node-local buffer called *Gather&Sort*, and the global shared sketch. Moreover, the shared sketch itself is organized in multiple levels, which may be propagated (and sorted) concurrently by multiple threads. While the sequential case may grow logarithmically with the stream size [5], the memory overhead of Quancurrent depends only on the number of update threads and on the number of NUMA nodes, and is independent of the stream size.

To allow queries to scale as well, Quancurrent serves them from a cached snapshot of the shared sketch. This architecture is illustrated in Figure 1. The query freshness depends on the sizes of local and NUMA-local buffers as well as the frequency of caching queries. We show that using this architecture, high throughput can be achieved with much smaller buffers (hence much better freshness) than in FCDS.

The collaborative propagation of data to and from shared buffers creates a synchronization challenge. In order to reduce synchronization overhead, we accommodate occasional data races. Specifically, we allow buffered elements to be sporadically overwritten by others without being propagated, and others to be duplicated, i.e., propagated more than once. These occurrences, which we call *holes*, may result in correlated sampling. When an element is duplicated, we sample the same element twice instead of sampling an independent element. Note that holes are randomly sampled from the same distribution as the original stream. Thus, they have no effect on the sampling mean. Nevertheless, the correlated sampling may increase the variance and thus affect the accuracy of estimation. We show in two ways that the error holes introduce is negligible. First, we empirically show that holes have a marginal effect on accuracy. Figure 2 presents quantiles estimated by Quancurrent on a stream of normally distributed random values (depicted as red circles) compared to an exact, brute-force computation of the quantiles (green dots). We can see that the estimation is accurate. Second, in Section 4 we statistically analyze the expected number of holes under the assumption of a *uniform stochastic scheduler* [6]. We show that the probability for a hole is negligible.

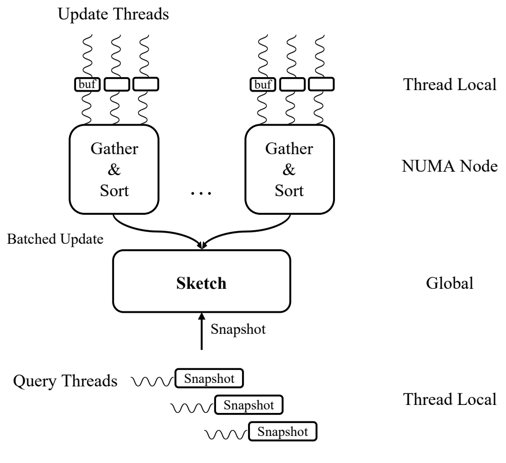


**Figure 1: Quancurrent's data structures.**

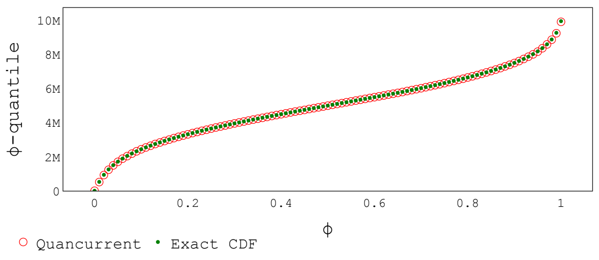


**Figure 2: Quancurrent quantiles vs. exact CDF, k = 1024, normal distribution, 32 update threads, 10M elements.**

In Section 5 we empirically evaluate Quancurrent on a 32-core system. We observe linear speedup over the sequential sketch of both queries and updates, peaking at 12x for update-only and 30x for queries concurrent with updates. We compare Quancurrent to FCDS, which is the state-of-the-art in concurrent sketches, and show that for FCDS to achieve similar performance it requires an order of magnitude larger buffers than Quancurrent, reducing query freshness tenfold.

In the Appendix we formally define the system model and present formal correctness proofs.

## ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. nnnnnnn and Grant No. mmmmmmm. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

## 2 BACKGROUND

### 2.1 Problem Definition

Given a stream $A = x_1, x_2, \ldots, x_n$ with $n$ elements, the *rank* of some $x$ (not necessarily in $A$) is the number of elements smaller than $x$ in $A$, denoted $R(A, x)$. For any $0 \leq \phi \leq 1$, the $\phi$ *quantile* of $A$ is an element $x$ such that $R(A, x) = \lfloor \phi n \rfloor$.

A Quantiles sketch's API is as follows:

- **update($x$)** process stream element $x$;
- **query($\phi$)** return an approximation of the $\phi$ quantile in the stream processed so far.

A PAC Quantiles sketch with parameters $\epsilon, \delta$ returns element $x$ for query($\phi$) after n updates such that $R(A, x) \in [(\phi - \epsilon)n, (\phi + \epsilon)n]$, with probability at least $1 - \delta$.

In an $r$-relaxed sketch for some $r \geq 0$ every query returns an estimate of the $\phi$ quantile in a subset of the stream processed so far including all but at most $r$ stream elements [15, 19].

## 2.2 Sequential Implementation

The Quantiles sketch proposed by Agarwal et al. [5] consists of a hierarchy of arrays, where each array summarizes a subset of the overall stream. The sketch is instantiated with a parameter $k$, which is a function of $(\epsilon, \delta)$. The first array, denoted level 0, consists of at most $2k$ elements, and every subsequent array, in levels $1, 2, \ldots,$ consists of either 0 or $k$ elements at any given time.

Stream elements are processed in order of arrival, first entering level 0, until it consists of $2k$ elements. Once this level is full, the sketch samples the array by sorting it and then selecting either the odd indices or the even ones with equal probability. The $k$ sampled elements are then propagated to the next level, and the rest are discarded. If the next level is full, i.e., consists of $k$ elements, then the sketch samples the union of both arrays by performing a merge sort, and once again retaining either the odd or even indices with equal probability. This propagation is repeated until an empty level is reached. Every level that is sampled during the propagation is emptied. Figure 3 depicts the processing of $4k$ elements.

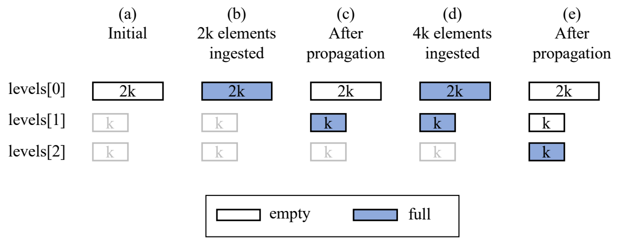


**Figure 3: Quantiles sketch structure and propagation.**

Each element is associated with a *weight*, which is the number of coin flips it has "survived". An element in an array on level $i$ has a weight of $2^i$, as it was sampled $i$ times. Thus, an element with a weight of $2^i$ represents $2^i$ elements in the processed stream. For approximating the $\phi$ quantile, we construct a list of tuples, denoted *samples*, containing all elements in the sketch and their associated weights. The list is then sorted by the elements' values. Denote by $W(x_i)$ the sum of weights up to element $x_i$ in the sorted list. The estimation of the $\phi$ quantile is an element $x_j$, such that $W(x_j) \leq \lfloor \phi n \rfloor$ and $W(x_{j+1}) > \lfloor \phi n \rfloor$.

## 3 QUANCURRENT

We present Quancurrent, an $r$-relaxed concurrent Quantiles sketch where $r$ depends on system parameters as discussed below. The algorithm uses $N$ update threads to ingest stream elements and allows an unbounded number of query threads. Queries are processed at any time during the sketch's construction. We consider a shared memory model that provides synchronization variables (atomics) and atomic operations to guarantee sequential consistency as in C++ [7]. Everything that happened before a write in one thread becomes visible to a thread that reads the written value. Also, there is a single total order of writes that all threads observe. We use the following sequentially consistent atomic operations (which force a full fence): *fetch-and-add (F&A)* [3] and *compare-and-swap (CAS)* [2].

In addition, we use a software-implemented higher-level primitive, *double-compare-double-swap (DCAS)* which atomically updates two memory addresses as follows: DCAS($addr_1$: $old_1 \rightarrow new_1$, $addr_2$: $old_2 \rightarrow new_2$) is given two memory addresses $addr_1$, $addr_2$, two corresponding expected values $old_1$, $old_2$, and two new values $new_1$, $new_2$ as arguments. It atomically sets $addr_1$ to $new_1$ and $addr_2$ to $new_2$ only if both addresses match their expected values, i.e., the value at $addr_1$ equals $old_1$ and the value at $addr_2$ equals $old_2$. DCAS also provides wait-free DCAS_READ primitive, which can read fields that are concurrently modified by a DCAS. DCAS can be efficiently implemented using single-word CAS [13, 14] in case the parameters take one word each. Otherwise, it can be implemented using locks.

In Section 3.1, we present the data structures used by Quancurrent. Section 3.2 presents the update operation, and Section 3.3 presents the query. The formal correctness proof is deferred to the supplementary material.

## 3.1 Data Structures

Quancurrent's data structures are described in Algorithm 1 and depicted in Figure 4. Similarly to the sequential Quantiles sketch, Quancurrent is organized as a hierarchy of arrays called *levels* where the number of levels grows logarithmically with the stream size. For convenience, we use a parameter *MAX_LEVEL* to describe the maximum number of levels. In principle, there is no limit on the number of levels, though particular implementation can limit it. Each level can be *empty*, *full*, or in *propagation*. The variable *tritmap* maintains the states of all levels. Tritmap is an unsigned integer, interpreted as an array of trits (trinary digits). The trit $tritmap[i]$ describes level $i$'s state: if $tritmap[i]$ is 0, level $i$ contains 0 or $2k$ ignored elements and is considered to be empty. If $tritmap[i]$ is 1, level $i$ contains $k$ elements and is deemed full, and if it is 2, level $i$ contains $2k$ elements and is associated with the propagation state. Each thread has a local buffer of size $b$, *localBuf* $[b]$. Before being ingested into the sketch's levels, stream elements are buffered in threads' local buffers and then moved to a processing unit called *Gather&Sort*. The *Gather&Sort* object has two $2k$-sized shared buffers, *G&SBuffer*$[2]$, each with its own *index* specifying the current location, as depicted in Figure 4a.

The query mechanism of Quancurrent includes taking an atomic snapshot of the levels. Query threads cache the snapshot and the tritmap that represents it in local variables, *snapshot* and *myTrit*, respectively. As the snapshot reflects only the sketch's levels and not G&SBuffers or the thread's local buffers, Quancurrent is $(4kS + (N-S)b)$-relaxed Quantiles sketch where $N$ is the number of update threads and $S$ is the number of NUMA nodes.

## 3.2 Update

The ingestion of stream elements occurs in three stages: (1) *gather and sort*, (2) *batch update*, and (3) *propagate level*. In stage (1), stream elements are buffered and sorted into batches of $2k$ through a *Gather&Sort* object. Each NUMA node has its designated *Gather&Sort* object, which is accessed by NUMA-local threads. Stage (2) executes a batch update of $2k$ elements from the *Gather&Sort* object to $levels[0]$. Finally, in stage (3), $levels[0]$ is propagated up the levels of the hierarchy.

**Algorithm 1:** Quancurrent data structures

```
1  Parameters and constants:
2      $MAX\_LEVEL$
3      $k$                                  ▷ sketch level size
4      $b$                                  ▷ local buffer size
5      $S$                                  ▷ #NUMA nodes
6  Shared objects:
7      $tritmap \leftarrow 0$
8      $levels[MAX\_LEVEL]$
9  NUMA-local objects:
                         ▷ shared among threads on the same node
10     $G\&SBuffer[2][2k]$
11     $index[2] \leftarrow \{0, 0\}$
12 Thread local objects:
13     $localBuf[b]$
14     $myTrit$                             ▷ used by query
15     $snapshot$                           ▷ used by query
```

In the first stage, threads first process stream elements into a thread-local buffer of size $b$. Once the buffer is full, it is sorted and the thread reserves $b$ slots on a shared buffer in its node's Gather&Sort unit. It then begins to move the local buffer's content to the shared buffer. The shared Gather&Sort buffer contains $2k$ elements, and its propagation (during Stage 2) is not synchronized with the insertion of elements. Thus, some reserved slots might still contain old values, (which have already been propagated), instead of new ones. As the batch is a sample of the original stream, we can accept the possible loss of information in order to improve performance. Below, we show that the sampling bias this introduces is negligible.

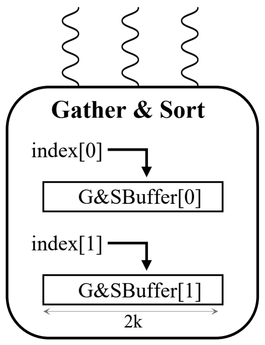
The pseudo-code for the first stage is presented in Algorithm 2. To insert its elements to the shared buffer, a thread tries to reserve $b$ places in one of the shared buffers using F&A (Line 22). If the index does not overflow, the thread copies its local buffer to the reserved slots (Line 24). We refer to the thread that fills the last $b$ locations in a G&SBuffer as the *owner* of the current batch. The batch owner creates a locally sorted copy of the shared buffer and begins its propagation (Lines 26-27). As each update thread sorts its local buffer before moving it to G&SBuffer, a full G&SBuffer consists of $\frac{2k}{b}$ sorted segments (referred to as *regions*) of size $b$. Thus, the owner simply performs a *b-way merge* to sort the shared buffer.

Note that the local buffer is not atomically moved into the shared buffer (Line 24 is a loop). Thus, the owner might begin a propagation before another thread has finished moving its elements to the shared buffer. In this case, the old elements already contained within the G&SBuffer are taken instead. Furthermore, upon moving its elements later, the writer thread might overwrite more recent elements. In other words, during this stage, stream elements may be duplicated and new elements may be dropped. We call these occurrences *holes* and analyze their implications in Section 4.1. Holes may cause some regions in the G&SBuffer to be unsorted. To address this, the owner thread creates a local copy of the shared buffer and then performs a single pass to validate that each region is monotonic, and if not, sorts it. Lastly, the owner uses a b-way merge to sort the full buffer.

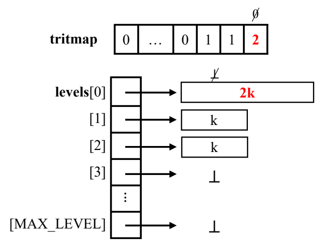
**Algorithm 2:** Stage 1: gather and sort

```
16 Procedure update($x$):
17     add $x$ to $localBuf$                        ▷ thread-local
18     if $\neg localBuf$.full() then return
19     sort $localBuf$
20     $i \leftarrow 0$
21     do                                ▷ insert to Gather&Sort unit
22         $idx \leftarrow index[i]$.F&A($b$)
23         if $idx < 2k$ then                     ▷ space available
24             move $localBuf$ to $G\&SBuffer[i][idx, \ldots, idx + b]$
25             if $idx + b = 2k$ then          ▷ owner, filled buffer
26                 $myCopy \leftarrow$ sorted copy of $G\&SBuffer[i]$
27                 batchUpdate($i$,$myCopy$)
28             return
29         $i \leftarrow \neg i$
30     while $true$
31 end
```

In the second stage, the owner inserts its local sorted copy of the shared buffer into level 0 using a DCAS. The batch of $2k$ elements is only inserted when level 0 is empty, reflected by the first digit of the tritmap being 0. We use DCAS to atomically update both *levels*[0] to point to the new sorted batch and *tritmap* to indicate an ongoing batch update (reflected by setting *tritmap*[*0*] to 2). The



**(a)** *Gather&Sort* **object.**



**(b) Batch update into** *levels*[0]**.**

**Figure 4: Quancurrent's data structures.**

DCAS might fail if other owner threads are trying to insert their batches or propagate them. The owner keeps trying to insert its batch into the sketch's first level until a DCAS succeeds, and then resets the index of the G&SBuffer to allow other threads to ingest new stream elements. The pseudo-code for the second stage is presented in Algorithm 3, and an example is depicted in Figure 4b.

**Algorithm 3:** Stage 2: batch update

```
32 Procedure batchUpdate(i,base_copy):
33     while ¬DCAS($levels[0]$: $\perp \longrightarrow$ $base\_copy$, $tritmap[0]$: 0
         $\longrightarrow$ 2 ) do { }
34     $index[i] \longleftarrow 0$
35     propagate(0)
36 end
```

In the beginning of the third stage, level 0 points to a new sorted copy of a *G&SBuffer* array and *tritmap*[0]=2. During this stage, the owner thread propagates the newly inserted elements up the levels hierarchy iteratively, level by level from level 0 until an empty level is reached. The pseudo-code for the propagation stage is presented in Algorithm 4. On each call to *propagate*, level $l$ is propagated to level $l+1$, assuming that level $l$ contains $2k$ sorted elements and $tritmap[l] = 2$. If $tritmap[l+1] = 2$, the owner thread is blocked by another propagation from $l+1$ to $l+2$ and it waits until $tritmap[l+1]$ is either a 0 or 1. The owner thread samples $k$ elements from level $l$ and retains the odd or even elements with equal probability (Line 39). If $tritmap[l+1]$ is 1, then level $l+1$ contains $k$ elements. The sampled elements are merged with level $l$+1 elements into a new $2k$-sized sorted array (Line 41). We then (in Line 42) continuously try, using DCAS, to update $levels[l+1]$ to point to the merged array and atomically update *tritmap* such that $tritmap[l] \longleftarrow 0$, reflecting level $l$ is available, and $tritmap[l+1] \longleftarrow 2$, reflecting that level $l$+1 contains $2k$ elements. That is, DCAS takes 3 arguments for each update word (Double), for the first word, the address of $levels[l+1]$, its expected value i.e., $levels[l+1]$, and the new merged array, and for the second word, the address of **all** the *tritmap* variable, its **full** expected value, and the **full** new value. For readability, in the DCAS pseudo-code, we emphasize the specific part being updated (for example, Line 42). After a successful DCAS, we clear level $l$ (set it to $\perp$) and proceed to propagate the next level (Line 44). If $tritmap[l+1]$ is 0, then level $l$+1 is empty. We use DCAS (Line 45) to update $levels[l+1]$ to point to the sampled elements and atomically update *tritmap* so that $tritmap[l]$ becomes 0, and $tritmap[l+1]$ becomes 1 (containing $k$ elements). After a successful DCAS, we clear level $l$ (set it to $\perp$) and end the current propagation.

Propagations of different batches may occur concurrently, i.e., level propagation of levels $l$ and $l'$ can be performed in parallel. Figure 5 depicts an example of concurrent propagation of two batches.

## 3.3 Query

Queries are performed by an unbounded number of query threads. A query returns an approximation based on a subset of the stream processed so far including all elements whose propagation into the levels array began before the query was invoked. The query is served from an atomic snapshot of the levels array. The pseudo-code is presented in Algorithm 5. Instead of collecting a new snapshot for each query, we cache the snapshot so that queries may be serviced from this cache, as long as the snapshot is not too stale. The snapshot and the tritmap value that represents it are cached in local variables, *snapshot* and *myTrit*, respectively. Query freshness is controlled by the parameter $\rho$, which bounds the ratio between the current stream size and the cached stream size. As long as this threshold is not exceeded, the cached snapshot may be returned (Lines 50-51). Otherwise, a new snapshot is taken and cached.

**Algorithm 4:** Stage 3: Propagation of level $l$

```
37 Procedure propagate($l$):
38     if $l \geq MAX\_LEVEL$ then return
       ▷ choose odd or even indexed elements randomly
39     $newLevel \longleftarrow$ sampleOddOrEven($levels[l]$)
40     if $tritmap[l+1] = 1$ then                ▷ next level is full
41         $newLevel \longleftarrow$ merge($newLevel$, $levels[l+1]$)
42         while ¬DCAS($levels[l+1]$: $levels[l+1] \longrightarrow newLevel$,
             $tritmap[l, l+1]$: $[2,1] \longrightarrow [0,2]$) do { }
43         $levels[l] \longleftarrow \perp$                          ▷ clear level
44         return propagate($l$+1)
       ▷ $tritmap[l+1]$ is 0 or 2
45     while ¬DCAS($levels[l+1]$: $\perp \longrightarrow newLevel$, $tritmap[l,$
         $l+1]$: $[2,0] \longrightarrow [0,1]$) do { }
46     $levels[l] \longleftarrow \perp$                              ▷ clear level
47 end
```

The snapshot is obtained by first reading the tritmap, then reading the levels from 0 to *MAX_LEVEL*, and then reading the tritmap again. If both reads of the tritmap represent the same stream size then they represent the same stream. We can use the levels read to reconstruct some state that represents this stream. The process is repeated until two such tritmap values are read. For example, focusing on the last two phases of the propagation in Figure 5, we assume a query thread $T_q$ reads $tm1 = 00202$, then reads the levels from $levels[0]$ to $levels[4]$ as depicted in Figure 5 (between the dashed lines), and then read $tm2 = 00210$. The two tritmap reads represent the same stream of size $10k$, thus a snapshot representing the same stream can be constructed from the levels read. The pseudo-code for calculating the stream size is presented in Algorithm 6. Note, queries are not wait-free and may starve in case of frequent updates. This can be addressed by blocking updates [25] but we found no practical need to do so since our experiments do not indicate any practical problem. Each level is read atomically as the levels' arrays are immutable and replaced by pointer swings. The snapshot is a subset of the levels summarizing the stream. To construct the snapshot, the collected levels are iterated over, in reversed order, from *MAX_LEVEL* to 0, and level $i$ is added to the snapshot only if the total collected stream size (including level $i$) is less than or equal to the stream size represented by the tritmap (Line 61). Back to our last example, the size of each level collected by $T_q$ is $2k, k, 2k, 0, 0$ (in descending order). As explained, to construct the snapshot, we go over the collected levels from $snapLevels[4]$ to $snapLevels[0]$. By reading $snapLevels[1]$, the total stream size represented by the

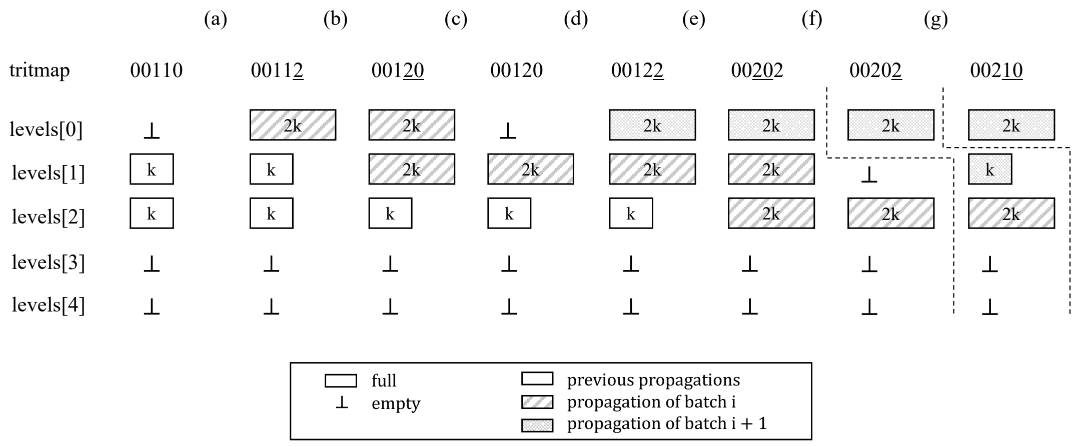


**Figure 5: Quancurrent propagation.**
**(a) The owner of batch $i$, owner($i$), inserts batch $i$ to level 0 and atomically updates $tritmap[0]$ to 2. (b) owner($i$) merges level 0 with level 1 and changes $tritmap[1,0]$ from $[1,2]$ to $[2,0]$. (c) owner($i$) clears level 0. (d) owner($i+1$) inserts its batch to level 0 and atomically updates $tritmap[0]$ to 2. (e) owner($i$) merges level 1 with level 2, and sets $tritmap[2,1]$ to $[2,0]$. Batch $i+1$ is still blocked because level 1 has not been cleared yet. (f) owner($i$) clears level 1. (g) Now owner($i+1$) successfully merges level 0 with the empty level 1, and sets $tritmap[1,0]$ to $[1,0]$.**

current snapshot is $0+0+4\cdot 2k+2\cdot k=10k$. As the stream size represented by $tm1$ and $tm2$ is $10k$, the construction of the snapshot is done and all elements of the processed stream are represented exactly once. The tritmap *myTrit* maintains the total size of the collected stream and each trit describes the state of a collected level. If level $i$ was collected to the snapshot, the value of *myTrit*$[i]$ is the size of level $i$ divided by $k$ (Line 63).

As levels propagate from lowest to highest, reading the levels in the same direction ensures that no element would be missed but may cause elements to be represented more than once. Building the snapshot from highest to lowest ensures that each element will be accounted once. In other words, reading the levels from lowest to highest and building the snapshot from highest to lowest ensures that an atomic snapshot is collected, as proven in the Supplementary material.

**Algorithm 5:** Query

```
48 Procedure Query($\phi$):
49     $tm1 \leftarrow$ tritmap
50     if $\frac{tm1.streamSize()}{myTrit.streamSize()} \le \rho$ then
51         return snapshot.query($\phi$)
52     do
53         $tm1 \leftarrow$ tritmap
54         $snapLevels \leftarrow$ read $levels$ 0 to $MAX\_LEVEL$
55         $tm2 \leftarrow tritmap$
56     while $tm1$.streamSize() $\ne$ $tm2$.streamSize()
57     $myTrit \leftarrow 0$
58     $snapshot \leftarrow$ empty $snapshot$
59     for $i \leftarrow MAX\_LEVEL$ to 0 do
60         $weight \leftarrow 2^i$
61         if $snapLevels[i]$.size()$\cdot weight+$
             $myTrit$.streamSize()$\le tm1$.streamSize() then
62             add $snapLevels[i]$ to $snapshot$
63             $myTrit[i] \leftarrow snapLevels[i]$.size()$/k$
64             if $myTrit$.streamSize()=$tm1$.streamSize() then
                 break
65     end
66     return $snapshot$.query($\phi$)
67 end
```

## 4 ANALYSIS

In Section 4.1 we analyze the expected number of holes, and in Section 4.2 we analyze Quancurrent's error.

### 4.1 Holes Analysis

Because the update operation moves elements from the thread's local buffer to a shared buffer non-atomically, holes may occur when the owner thread reads older elements that were written to the shared buffer in a previous window. The missed (delayed) writes may later overwrote newer writes. Together, for each hole, an old value is duplicated and a new value is dropped. As such, we created a dependency between samples because we dropped an independent sample and gave double weight to another.

We analyze the expected number of holes under the assumption of a *uniform stochastic scheduler* [6], which schedules each thread with a uniform probability in every step. That is, at each point in the execution, the probability for each thread to take the next step is $\frac{1}{N}$. Note that the holes are random and distributed from the same distribution. Therefore, they do not affect the samples' mean and only affect the accuracy of estimation. Below we show that the expected number of holes is fairly small and that they have a marginal effect on the estimation accuracy.

Denote by $H$ the total number of holes in some batch of $2k$ elements. G&SBuffer's array is divided into $\frac{2k}{b}$ regions, each consisting

**Algorithm 6:** Tritmap

```
Procedure streamSize():
    $curr\_stream \leftarrow 0$
    for $i \leftarrow 0$ to $MAX\_LEVEL$ do
        $weight \leftarrow 2^i$
        if $tritmap[i] = 1$ then
            $curr\_stream \leftarrow curr\_stream + weight \cdot k$
        else if $tritmap[i] = 2$ then
            $curr\_stream \leftarrow curr\_stream + weight \cdot 2k$
    end
    return $curr\_stream$
end
```

of $b$ slots populated by the same thread. Denote by $H_1, \ldots, H_{\frac{2k}{b}}$ the number of holes in regions $1, \ldots, \frac{2k}{b}$, respectively.

The slots in region $j$ are written to by the thread that successfully increments the shared index from $(j-1)b$ to $jb$. We refer to this thread as $T_j$. Note that multiple regions may have the same writing thread. The shared buffer's owner, $T_O$, is $T_{\frac{2k}{b}}$. To initiate a batch update, $T_O$ creates a local copy of one G&SBuffer by iteratively reading the array. A hole is read in some region $j$ if $T_O$ reads some index $i+1$ in this region before the writer thread $T_j$ writes to the corresponding index in the same region.

*Analysis of* $\mathbf{H_j}$. When $T_O$ increments the index from $2k-b$ to $2k$, $T_j$ may have completed any number of writes between 0 and $b$ to region $j$. We first consider the case that $T_j$ has not completed any writes. In this case, for a hole to be read in slot $i+1$ of region $j$, $T_O$'s read of slot $i+1$ must overtake $T_j$'s write of the same slot. To this end, $T_O$ must write $b$ values (from its own local buffer), read $(j-1)b$ values from the first $j-1$ regions and then read values from slots $1, \ldots, i+1$ in this region before $T_j$ takes $i+1$ steps. The probability that $T_O$ reads a hole for the first time in this region in slot $i+1$ is:

$$\pi_{i,j} \triangleq P[\text{hole in slot } i+1 \mid \text{no hole in slots } 1 \ldots i] \cdot P[\text{no hole in slots } 1 \ldots i].$$

For a hole to be read in slot $i+1$ of region $j$, $T_O$ must take $b + (j-1)b + i + 1$ steps while $T_j$ takes at most $i$ steps, with $T_O$'s read of slot $i+1$ being last. But if $T_j$ takes fewer than $i$ steps, a hole is necessarily read earlier than slot $i+1$. Therefore, we can bound $\pi_{i,j}$ by considering the probability that $T_j$ takes exactly $i$ steps while $T_O$ takes $b + (j-1)b + i$ steps, and then $T_O$ takes a step. Ignoring steps of other threads, each of $T_j$ and $T_O$ has a probability of $\frac{1}{2}$ to take a step before the other. Therefore,

$$\pi_{i,j} \leq \left(\frac{1}{2}\right)^{jb+2i+1} \binom{jb+2i}{i}.$$

Note that this includes schedules in which $T_O$ reads holes in previous slots in the same region, therefore it is an upper bound. Given that $T_j$ has not yet written in region $j$, the probability, $p_j$, that $T_O$ reads at least 1 hole in region $j$ is bounded as follows:

$$p_j \leq \sum_{i=0}^{b-1} \pi_{i,j}$$

If $T_j$ has completed writes to region $j$, the probability that $T_O$ reads holes is even lower. Therefore, the probability that $H_j \geq 1$ is bounded from above by $p_j$. Using this, we bound the expected total number of holes in region $j$:

$$E\left[H_j\right] = P(H_j = 0) \cdot 0 + P(H_j = 1) \cdot 1 + \cdots + P(H_j = b) \cdot b.$$

$T_O$ can read at most $b$ holes, therefore,

$$\begin{aligned} E\left[H_j\right] &< b \cdot \left(P(H_j = 1) + \cdots + P(H_j = b)\right) \\ &= b \cdot P(H_j \geq 1) < b \cdot p_j. \end{aligned}$$

Using the linearity of expectation, we bound the expected number of holes in a batch:

$$E\left[H\right] = E\left[H_1\right] + E\left[H_2\right] + \cdots + E\left[H_{\frac{2k}{b}}\right].$$

In the supplementary material, we prove that

$$\begin{aligned} &\forall j \geq 1, b \in \mathbb{N},\ E[H_{j+1}] \leq 0.5 \cdot E[H_j] \\ &\forall b \in \mathbb{N},\ E[H_1] \leq 1.4 \end{aligned}$$

Together, this implies that $E[H] \leq 2.8$ for all $b \in \mathbb{N}$.

## 4.2 Error Analysis

The source of Quancurrent's estimation error is twofold: (1) the error induced by sub-sampling the stream, and (2) the additional error induced by concurrency. For the former, we leverage the existing literature on analysis of sequential sketches. We analyze the latter. As the expected number of holes is fairly small and the holes are random, we disregard their effect on the error analysis.

First, our buffering mechanism induces a relaxation. Let $S$ be the number of NUMA nodes. Recall that each NUMA node has a Gather&Sort object that contains two buffers of size $2k$. In addition, each of the $N$ update threads has a local buffer. When the G&SBuffer is full, the local buffer of the owner is empty so at most $N-S$ threads locally buffered elements. Therefore, the buffering relaxation $r$ is $4kS + (N-S)b$.

Rinberg et al. [19] show that for a query of a $\phi$-quantile, an $r$-relaxation of a Quantiles sketch with parameters $\epsilon_c$ and $\delta_c$, returns an element whose rank is in the range $[(\phi - \epsilon_r)n, (\phi + \epsilon_r)n]$ with probability at least $1 - \delta_c$, for $\epsilon_r = \epsilon_c + \frac{r}{n}(1 - \epsilon_c)$.

On top of this relaxation, our cache mechanism induces further staleness. Here, the staleness depends on $\rho$. Let $n_{old}$ be the stream size of the cached snapshot, and let $n_{new}$ be the current stream size. If $n_{new}/n_{old} \leq \rho$ then the query is answered from the cached snapshot. Denote $\rho \triangleq 1 + \epsilon'$ for some $\epsilon' \geq 0$. The element returned by the cached snapshot is in the range:

$$\left[\left(\phi - \epsilon_r\right) n_{old}, \left(\phi + \epsilon_r\right) n_{old}\right]$$

As $n_{old} \leq n_{new}$, then,

$$\left(\phi + \epsilon_r\right) n_{old} \leq \left(\phi + \epsilon_r\right) n_{new} \leq \left(\phi + \left(\epsilon' + \epsilon_r\right)\right) n_{new}$$

On the other hand,

$$(\phi - \epsilon_r)\, n_{old} \geq (\phi - \epsilon_r)\, \frac{n_{new}}{\rho} =$$
$$\left(\frac{\phi}{1+\epsilon'} - \frac{\epsilon_r}{1+\epsilon'}\right) n_{new} =$$
$$\left(\phi - \frac{\phi\epsilon'}{1+\epsilon'} - \frac{\epsilon_r}{1+\epsilon'}\right) n_{new} \geq$$
$$\left(\phi - \left(\epsilon' + \epsilon_r\right)\right) n_{new}$$

Because $\phi \leq 1$ and $\epsilon' \geq 0$ then, $\frac{\phi\epsilon'}{1+\epsilon'} \leq \frac{\epsilon'}{1+\epsilon'} \leq 1$.
Therefore, the query returns a value within the range

$$[(\phi - \epsilon)\, n, (\phi + \epsilon)\, n]$$

for $\epsilon \triangleq \epsilon_r + \epsilon'$.

## 5 IMPLEMENTATION AND EVALUATION

In this section we measure Quancurrent's throughput and estimation accuracy. Section 5.1 presents the experiment setup and methodology. Section 5.2 presents throughput measurements and discusses scalability. Section 5.3 experiments with different parameter settings, examining how performance is affected by query freshness. Section 5.4 presents an accuracy of estimation analysis. Finally, Section 5.5 compares Quancurrent to the state-of-the-art.

### 5.1 Setup and Methodology

We implement Quancurrent in C++. In this paper, we implement the atomic DCAS using single-word CAS [13, 14]. We set *MAX_LEVEL* = 31 and *tritmap* is a single 64-bit word that can represent up to 31 levels, and summarize streams of up to $2^{31}k$ elements.

Our memory management system is based on IBR [24], an interval-based approach to memory reclamation for concurrent data structures. The experiments were run on a NUMA system with four Intel Xeon E5-4650 processors, each with 8 cores, for a total of 32 threads (with hyper-threading disabled).

Each thread was pinned to a NUMA node, and nodes were first filled before overflowing to other NUMA nodes, i.e., 8 threads use only a single node, while 9 use two nodes with 8 threads on one and 1 on the second. The default memory allocation policy is local allocation, except for Quancurrent's shared pointers. Each Gather&Sort unit is allocated on a different NUMA node and threads update the G&SBuffers allocated on the node they belong to. The stream is drawn from a uniform distribution unless stated otherwise. Each data point is an average of 15 runs, to minimize measurement noise.

### 5.2 Throughput Scalability

We measured Quancurrent's throughput in three workloads: (1) update-only, (2) query-only, and (3) mixed update-query. In the update-only workload, we update Quancurrent with a stream of 10M elements and measure the time it takes to feed the sketch. For the other two workloads, we pre-fill the sketch with a stream of 10M elements and then execute the workload (10M queries only or queries and 10M updates) and measure performance. Figure 6 shows Quancurrent's throughput in those workloads with $k = 4096$ and $b = 16$,

As shown in Figure 6a, Quancurrent's performance in the update-only workload with a single thread is similar to the sequential algorithm and with more threads it scales linearly, reaching $12x$ the sequential throughput with 32 threads. We observe that the speedup is faster with fewer threads, we believe this is because once there are more than 8 threads, the shared object is accessed from multiple NUMA nodes.

Figure 6b shows that, as expected, the throughput of the query-only workload scales linearly with the number of query threads, reaching $30x$ the sequential throughput with 32 threads.

In the mixed workload, the parameter $\rho$ is significant for performance - when $\rho = 1$ ($\epsilon' = 0$, no caching), a snapshot is reproduced on every query. Figure 6c presents the update throughput (left) and query throughput (right) in the presence of 1 or 2 update threads, with staleness thresholds of $\rho = 1$ ($\epsilon' = 0$) and $\rho = 1.05$ ($\epsilon' = 0.05$). We see that the caching mechanism ($\rho > 1$) is indeed crucial for performance. As expected, increasing the staleness threshold allows queries to use their local (possibly stale) snapshot, servicing queries faster and greatly increasing the query throughout. Furthermore, more update threads decrease the query throughput, as the update threads interfere with the query snapshot. Finally, increasing the number of query threads decreases the update throughput, as query threads interfere with update threads, presumably due to cache invalidations of the shared state.

### 5.3 Parameter Exploration

We now experiment with different parameter settings with up to 32 threads. In Figure 7a we vary $k$ from 256 to 4096, in update-only scenario with $b = 16$ and up to 32 update threads. We see that the scalability trends are similar, and that Quancurrent's throughput increases with $k$, peaking at $k = 2048$, after which increasing $k$ has little effect. This illustrates the tradeoff between the sketch size (memory footprint) to throughput and accuracy.

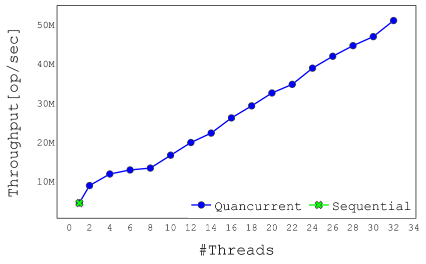
Figure 7b experiments with different local buffer sizes, from 1 to 64, in an update-only scenario with $k = 4096$ and up to 32 update threads. Not surprisingly, the throughput increases as the local buffer grow as this enables more concurrency.

In Figure 7c we vary $\rho$, in a mixed update-query workload with 8 update threads, 24 query threads, $k = 1024$, and $b = 16$, exploring another aspect of query freshness versus performance. As expected, increasing $\rho$ has a positive impact on query throughput, as the cached snapshot can be queried more often. Figure 7c also shows the miss rate, which is the percentage of queries that need to reconstruct the snapshot.

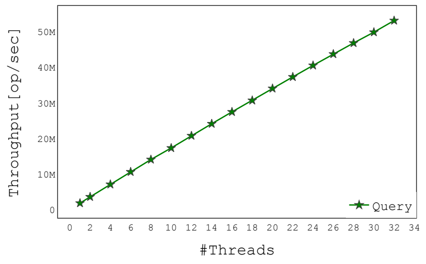
### 5.4 Accuracy

To measure the estimate accuracy, we consider a query invoked in a quiescent state where no updates occur concurrently with the query. Figure 8 shows the standard error of 1M estimations in a quiescent state. We see that Quancurrent's estimations are similar to the sequential ones using the same $k$, and improves with larger values of $k$ as known from the literature on sequential sketches [5].

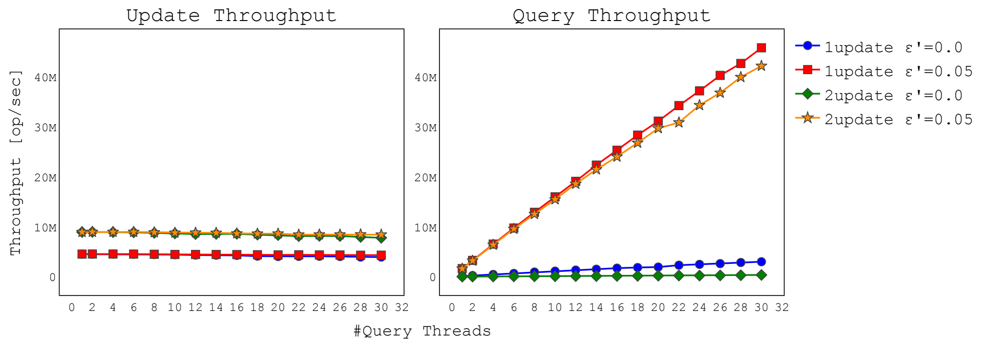
To illustrate the impact of $k$ visually, Figure 9 compares the distribution measured by Quancurrent (red open-circles) to the exact (full information) stream distribution (green CDF filled-circles). In Figure 2 (in the introduction), we depict the accuracy of Quancurrent's estimate of a normal distribution with $k = 1024$. Figure 9b



(a) Update-only, 10M elements.

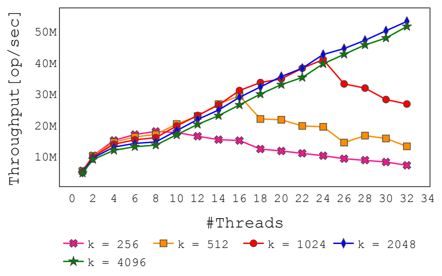


(b) Query-only, 10M elements prefilled, 10M queries.

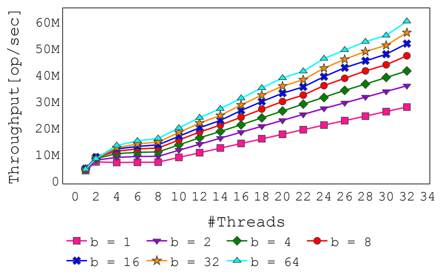


(c) One or two update threads, up to 30 query threads, 10M elements inserted after a pre-fill of 10M elements.

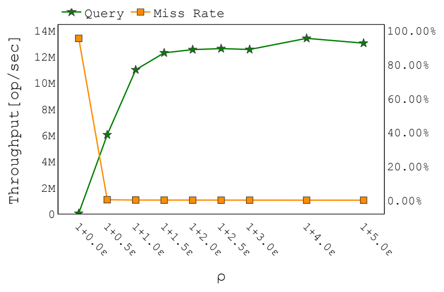
**Figure 6: Quancurrent throughput, k=4096, b=16.**



(a) Update-only, #keys=10M, b=16.



(b) Update-only, #keys=10M, k=4096.



(c) 8 update threads, 24 query threads, #keys=10M, k=1024 and b=16.

**Figure 7: Quancurrent parameters impact.**

(left) shows that when we reduce $k$ to 32, the approximation is less tight while for $k = 256$ (Figure 9b right) it is very accurate. We observe similar results for the uniform distribution in Figure 9a. We experimented with additional distributions with similar results, which are omitted due to space limitations.

## 5.5 Comparison with the state of the art

Finally, we compare Quancurrent against a concurrent Quantiles sketch implemented within the FCDS framework [19], the only previously suggested concurrent sketch we know that supports quantiles. Figure 10 shows the throughput results (log scale) for 8, 16, 24 and 32 threads and $k = 4096$. FCDS satisfies relaxed consistency with a relaxation of up to $2NB$, where $N$ is the number of worker threads and $B$ is the buffer size of each worker. Recall that Quancurrent's relaxation is at most $r = 4kS + (N - S)b$. Thus:

$$r_{\text{FCDS}} = 2NB \tag{1}$$

$$r_{\text{Quancurrent}} = 4kS + (N - S)b \tag{2}$$

For a fair comparison, we compare the two algorithms in settings with the same relaxation, as follows:

$$r_{\text{FCDS}} = r_{\text{Quancurrent}} \tag{3}$$

$$2NB = 4kS + (N - S)b \tag{4}$$

$$B = \frac{4kS + (N - S)b}{2N} \tag{5}$$

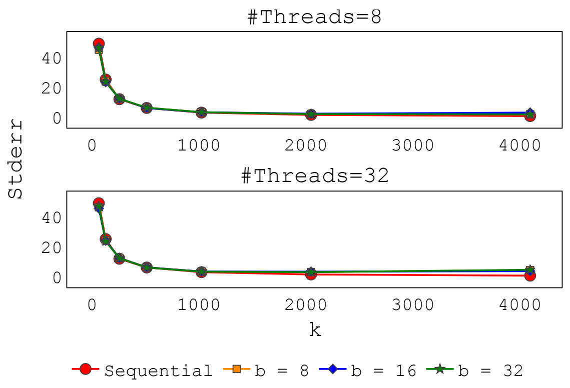
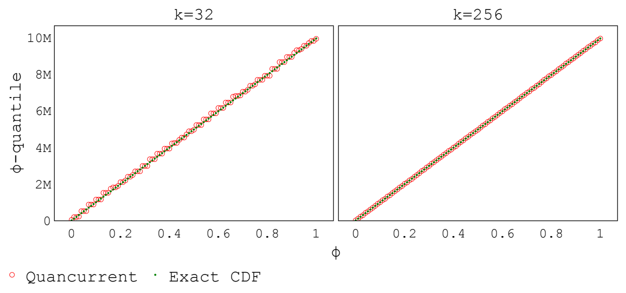
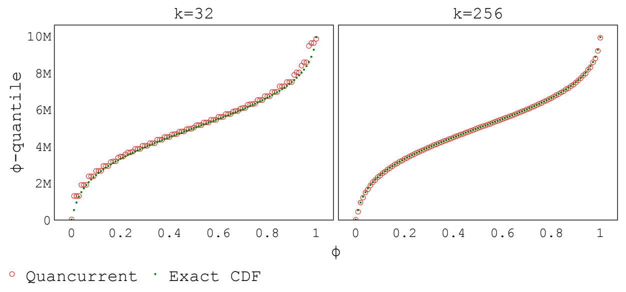


**Figure 8: Standard error of estimation in quiescent state, keys=1M, runs=1000.**



**(a) Uniform distribution.**



**(b) Normal distribution.**

**Figure 9: Quancurrent quantiles vs. exact CDF, with 32 threads, b=16, and a stream size of 10M.**

Given the sketch parameter $k$, the number of update threads $N$, the number of NUMA nodes $S$, and a series of Quancurrent' local buffer size $b$, we calculate the corresponding $B$ value (FCDS's local buffer size) using Equation 5. Note that the size of the local buffer $b$ is bounded by the size of the *G&SBuffer* array, which is $2k$.

The throughput results are shown in Figure 10. For clarity, some points with the same relaxation are highlighted using the same color in both curves. For 8 update threads ($S = 1$) and $b = 2048$, the relaxation of Quancurrent is $r \approx 30K$. The same relaxation in FCDS with the same number of update threads is achieved with a buffer size of $B = 1920$. With 8 threads, Quancurrent reaches a throughput of $22M$ $ops/sec$ for a relaxation of $30K$ whereas FCDS reaches a throughput of $25.8M$ $ops/sec$ for a much larger relaxation of $131K$. With 32 threads, Quancurrent reaches a throughput of $62M$ $ops/sec$ for a relaxation of $123K$, but FCDS only reaches a throughput of $19.4M$ $ops/sec$ with a relaxation of more than $500K$.

Overall, we see that FCDS requires large buffers (resulting in a high relaxation and low query freshness) in order to scale with the number of threads. This is because, unlike Quancurrent, FCDS uses a single thread to propagate data from all other threads' local buffers into the shared sketch. The propagation involves a heavy merge-sort, so large local buffers are required in order to offset it and keep the working threads busy during the propagation. In contrast, Quancurrent's propagation is collaborative, with merge-sorts occurring concurrently both at the NUMA node level (in Gather&Sort buffers) and at multiple levels of the shared sketch.

## 6 CONCLUSION

We presented Quancurrent, a concurrent scalable Quantiles sketch. We have evaluated it and shown it to be linearly scalable for both updates and queries while providing accurate estimates. Moreover, it achieves higher performance than state-of-the-art concurrent quantiles solutions with better query freshness. Quancurrent's scalability arises from allowing multiple threads to concurrently engage in merge-sorts, which are a sequential bottleneck in previous solutions. We dramatically reduce the synchronization overhead by accommodating occasional data races that cause samples to be duplicated or dropped, a phenomenon we refer to as holes. This approach leverages the observation that sketches are approximate to begin with, and so the impact of such holes is marginal. Future work may leverage this observation to achieve high scalability in other sketches or approximation algorithms.

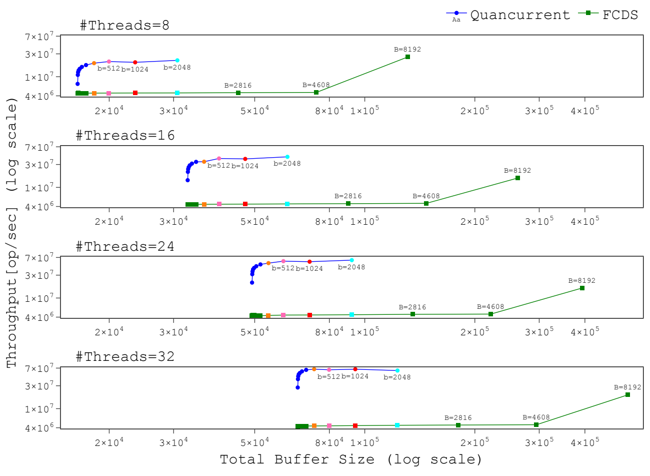


**Figure 10: Quancurrent vs. FCDS, k = 4096.**

## REFERENCES

[1] 2019. Apache DataSketches. https://datasketches.apache.org/.

[2] Accessed: March 2022. Compare and Exchange. https://c9x.me/x86/html/file_module_x86_id_41.html.

[3] Accessed: March 2022. Exchange and Add. https://c9x.me/x86/html/file_module_x86_id_327.html.

[4] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. 2013. Scuba: Diving into data at facebook. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1057–1067.

[5] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. 2012. Mergeable Summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Scottsdale, Arizona, USA) *(PODS '12)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/2213556.2213562

[6] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. 2016. Are lock-free concurrent algorithms practically wait-free? *Journal of the ACM (JACM)* 63, 4 (2016), 1–20.

[7] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

[8] Mihai Budiu, Parikshit Gopalan, Lalith Suresh, Udi Wieder, Han Kruiger, and Marcos K Aguilera. 2019. Hillview: a trillion-cell spreadsheet for big data. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1442–1457.

[9] Graham Cormode, Zohar Karnin, Edo Liberty, Justin Thaler, and Pavel Veselỳ. 2021. Relative error streaming quantiles. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 96–108.

[10] Lee Rhodes Daniel Ting, Jonathan Malkin. 2020. Data Sketching for Real Time Analytics: Theory and Practice. https://datasketches.apache.org/docs/Community/KDD_Tutorial_Summary.html.

[11] Druid. Accessed February 16, 2022. Apache Druid. https://druid.apache.org/docs/latest/development/extensions-core/datasketches-quantiles.html.

[12] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. 2018. Moment-based quantile sketches for efficient high cardinality aggregation queries. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1647–1660.

[13] Rachid Guerraoui, Alex Kogan, Virendra J Marathe, and Igor Zablotchi. 2020. Efficient Multi-Word Compare and Swap. In *34th International Symposium on Distributed Computing*.

[14] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A Practical Multi-Word Compare-and-Swap Operation. In *Proceedings of the 16th International Conference on Distributed Computing (DISC '02)*. Springer-Verlag, Berlin, Heidelberg, 265–279.

[15] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative Relaxation of Concurrent Data Structures. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2429069.2429109

[16] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal Quantile Approximation in Streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 71–78. https://doi.org/10.1109/FOCS.2016.17

[17] Charles Masson, Jee E Rim, and Homin K Lee. 2019. DDSketch: a fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2195–2205.

[18] Presto. Accessed February 16, 2022. PrestoDB. https://prestodb.io/docs/current/functions/aggregate.html.

[19] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. 2020. *Fast Concurrent Data Sketches*. Association for Computing Machinery, New York, NY, USA, 117–129. https://doi.org/10.1145/3332466.3374512

[20] Spark. Accessed February 16, 2022. Apache Spark. https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.DataFrame.approxQuantile.html.

[21] Charalampos Stylianopoulos, Ivan Walulya, Magnus Almgren, Olaf Landsiedel, and Marina Papatriantafilou. 2020. Delegation sketch: a parallel design with support for fast and accurate concurrent operations. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–16.

[22] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. 2015. Seedb: Efficient data-driven visualization recommendations to support visual analytics. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, Vol. 8. NIH Public Access, 2182.

[23] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. 2013. Quantiles over Data Streams: An Experimental Study. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 737–748. https://doi.org/10.1145/2463676.2465312

[24] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H. Alan Beadle, and Michael L. Scott. 2018. Interval-Based Memory Reclamation. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3178487.3178488

[25] Lior Zeno, Dan R. K. Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. 2022. SwiSh: Distributed Shared State Abstractions for Programmable Switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 171–191. https://www.usenix.org/conference/nsdi22/presentation/zeno

[26] Fuheng Zhao, Sujaya Maiyya, Ryan Wiener, Divyakant Agrawal, and Amr El Abbadi. 2021. KLL± Approximate Quantile Sketches over Dynamic Datasets. *Proc. VLDB Endow.* 14, 7 (apr 2021), 1215–1227. https://doi.org/10.14778/3450980.3450990