Quancurrent:
A Concurrent Quantiles Sketch

Shaked Elias Zada

Quancurrent:
A Concurrent Quantiles Sketch

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering

Shaked Elias Zada

Submitted to the Senate
of the Technion — Israel Institute of Technology Elul,
5782 Haifa September 2022

This research was carried out under the supervision of Prof. Idit Keidar, in the Faculty of Electrical and Computer Engineering.

Acknowledgements

First and foremost, I would like to thank the Dean and Professor Idit Keidar for being my Master's Thesis Supervisor. It was an honor and a privilege to work under your guidance. Thank you for believing in me and giving me the chance to learn from you, it has been an incredible and invaluable journey. As such, I am equally grateful to my fellow colleague and researcher Arik Rinberg. Thank you for being the constant anchor, to whom I could always turn to for ideas when I was stuck or simply to converse and learn from, your help made all the difference. To the members of Professor Idit's research group: Gal Assa, Oded Naor, Shir Cohen, and Ramy Fakhoury – your time, presence, insight, and constant support were indispensable, thank you for having my back. I would like to thank my family and friends for their love and continuous support. Especially, I would like to thank my husband Sagi for the endless amount of support, love, and encouragement to complete my academic journey. Thank you for always showing interest in my work and helping me get the best out of myself. Lastly, I couldn't have done this without my parents. I am forever grateful for their constant support, love, and belief in me.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

Abstract

Real-time analytics of large data streams is a fundamental task in the world of big data. When analyzing massive datasets, generating exact answers to even very basic queries about the data can require huge computing resources (memory and compute time). This leads to failure to scale as working with the full data is computationally infeasible.

Sketches are statistical data summaries of large data streams used to perform fast real-time analytics using only a small amount of memory space. They are a family of streaming algorithms processing massive datasets in a single pass. They can provide accurate, though approximate, answers with guaranteed error bounds to computationally complex queries orders of magnitude faster than traditional exact methods.

Describing the data distribution of a large stream of numeric values, such as web page load times, is a fundamental problem within the constraints of single-pass computation (the streaming model). We often desire to characterize the distribution of these values to understand the underlying trends or patterns in the data, for example, the median, fifth, and 95th percentile values. These are called quantiles. Quantiles are widely used representations for data distribution.

Thus, a popular sketch type is Quantiles, which estimates the data distribution of a large input stream. Quantiles sketches can estimate a quantile query up to a bounded error with bounded failure probability.

We present Quancurrent, a highly scalable concurrent Quantiles sketch that retains a small error bound with reasonable query freshness. 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 Quantiles sketch. Quancurrent allows queries to occur concurrently with updates and achieves an order-of-magnitude better query freshness than existing scalable solutions.

We prove our algorithm's correctness and analyze the approximation error and the query freshness.

Notation and Abbreviations

N	Number of update threads	17
S	Number of NUMA nodes	18
δ	Bound on the failure probability	14
ϵ	Bound on the estimation error	14
MAX_LEVEL	Maximum number of levels in the sketch	18
ρ	Threshold for rebuilding a snapshot	22
b	Size of threads' local buffer	18
k	The sketch summary size	14
$R(A, x)$	Given a stream A with n elements the rank of some x (not necessarily in A) is the number of elements smaller than x in A	14
API	Application Programming Interface	14
CAS	Compare-And-Swap (atomic operation which takes a memory address, an expected value and a new value, compares the actual value with the expected value and if they match, atomically swap the actual value with the new value)	17
DCAS	Double-Compare-Double-Swap (atomic operation which takes two addresses, two corresponding expected values and two new values as arguments, and atomically updates both addresses if they both match their expected value)	17
DCAS_READ	Wait-free read operation (can read fields that are concurrently modified by a DCAS)	17
F&A	Fetch-And-Add	17
FCDS	Fast Concurrent Data Sketches (framework proposed by Rinberg et al. [37] for building concurrent data sketches)	8
IBR	Interval-Based-Reclamation (an interval-based approach to memory management for concurrent data structures)	35
NUMA	Non-Uniform-Memory-Access (memory design used in multiprocessing where the memory access time depends on the memory location relative to the processor)	8

PAC	Probably Approximately Correct (A is a PAC learning algorithm if for given $\epsilon > 0$ and $\delta < 1$ A outputs a hypothesis h that has an average error less than or equal to ϵ on the samples with probability at least $1 - \delta$)	7

Chapter 1

Introduction

Data sketches, or *sketches* for short, are indispensable tools for performing analytics on high-rate, high-volume incoming data streams [14]. A case study made by Yahoo has shown the impact of sketches on data analytics. Flurry [18] is a mobile application real-time analytics platform, acquired by Yahoo in 2014. Using data sketches architecture in Flurry reduced the overall cost of the system considerably. Sketches led to a reduction in data processing times from days or hours to minutes and even seconds [15].

Sketches typically estimate some function of a large data stream, for example, the frequency of certain items, or how many unique items have appeared. Sketches are quantitative objects that support update and query operations, where the return value of a query is from a totally ordered set.

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.). Since summaries have a much smaller size, they typically answer queries approximately, and there is a trade-off between the size of the summary and the approximation error. Some sketches can be deterministic, although most sketches are probabilistic in their behavior and take advantage of various randomization techniques. Typical sketches are *probably approximately correct (PAC)*, estimating some aggregate quantity with an error of at most ϵ with a probability of at least $1 - \delta$ for some parameters ϵ and δ [3, 9, 37, 44].

Some sketches have a well-known mergeability property [3], which enables computing a sketch over a stream by merging sketches over sub-streams and merging them into a single sketch whose error is no greater than if it had processed the entire stream. The ever-increasing rates of incoming data create a strong demand for parallel stream processing [11, 26]. Previous works have exploited the mergeability property for distributed stream processing, devising solutions with a sequential bottleneck at the merge phase.

With the rise of big data, a fundamental task in data management and data analysis is to describe the distribution of the data. This is used in applications such as exploratory data analysis [41], operation monitoring [1], and more. If the distribution is known a priori, such as a normal or zipifian distribution, it can be described by the parameters of the distribution. This

is rarely the case in practice, which calls for describing the distribution nonparametrically. Quantiles are the most commonly used nonparametric representation for data distribution. In particular, a technique such as quantile approximation, a nonparametric representation, is widely used to characterize data distributions [42, 44].

This problem has attracted a lot of prior studies, from both the algorithms and the database communities [34, 31, 27, 3, 5]. Quantiles are of interest to both database implementers and users: for instance, they are a fundamental tool for query optimization [31], splitting of data in parallel database systems [35, 11, 26], and statistical data analysis [28, 13]. In the past years, quantile estimation has received particular attention in the data streaming model, i.e., the data elements arrived one by one in a streaming fashion, and the algorithm has limited memory to work with [34, 31, 32, 22, 20, 38, 12, 27, 3, 29, 44].

The quantiles correspond to the *cumulative distribution function (CDF)*, which yields the *probability density function (PDF)*. They are a generalization of the median. Given A, a multi-set of n elements is drawn from a totally ordered universe, the ϕ-quantile of A, for some $0 \leq \phi \leq 1$, is the element whose rank is $\lfloor \phi n \rfloor$ in A, where the rank of an element x is the number of elements in A smaller than x. Note, given enough space, the quantiles can be easily found by sorting the set A.

Yet, when the algorithm's memory is limited and significantly smaller than the size of the data set, it is not possible to compute the quantile precisely. This was formalized in a 1980 paper by Munro and Peterson [34]. They have shown that computing the median with p passes over A has to use $\Omega(n^{-p})$ space. Thus, computing the true median will require memory linear in the size of the set. An alternate and more practical approach to the problem is to approximate the quantiles, represented as ϵ approximation ϕ-quantile: For an error parameter $0 < \epsilon < 1$, the ϵ-approximate ϕ-quantile is any element between $(\phi - \epsilon)n$ and $(\phi + \epsilon)n$.

The Quantiles sketch family captures this task [33, 3, 19, 10]. The Quantiles sketch represents the quantiles distribution in a stream of n elements, such that for any $0 \leq \phi \leq 1$, a query for quantile ϕ returns an estimate of the $\lfloor n\phi \rfloor^{\text{th}}$ largest element. Due to the importance of quantiles approximation, Quantiles sketches are a part of many analytics platforms, e.g., Druid [16], Hillview [7], Presto [36], and Spark [39].

The deterministic ϵ-approximation ϕ-quantile algorithms take as input a quantile query ϕ and a precision value ϵ and output an element x such that the quantile of x is in the range $[\phi - \epsilon, \phi + \epsilon]$. An alternative approach proposes a family of randomized algorithms where the output answer x is within the $[\phi - \epsilon, \phi + \epsilon]$ range with a high probability [32, 29]. These algorithms provide guarantees by bounding the failure probability to at most δ such that the user has $1 - \delta$ confidence that the sketch's output is ϵ-approximation.

The first deterministic streaming algorithm for quantiles was proposed by Manku, Rajagoplan, and Lindsay [31], building on the prior work by Munro and Paterson [34]. This algorithm has space complexity $O(\frac{1}{\epsilon}\log^2(\epsilon n))$, meaning that using memory that grows poly-logarithmically in the stream size and inversely with the accuracy parameter ϵ, the quantiles can be estimated with precision ϵn. We refer to this algorithm as MRL. This result has since been improved by two groups: In 2001, Greenwald and Khanna [22] designed a determin-

istic comparison-based algorithm (referred to as the GK algorithm) and showed that it uses $O(\frac{1}{\epsilon}\log(\epsilon n))$ space in the worst case. Hung and Ting [27] showed an $\Omega(\frac{1}{\epsilon}log(\epsilon n))$ lower bound for these deterministic algorithms. In this category, the GK algorithm is generally considered to be the best but it is not known to be mergeable. In 2004, Shrivastava et al. [38] designed a deterministic, fixed-universe algorithm, called Q-digest, that uses $O(\frac{1}{\epsilon}\log(u))$ space, where $[u]$ is a fixed finite universe from which the elements are drawn, meaning that the universe has to be known. This algorithm was designed for quantile computation in sensor networks and is a mergeable summary [3].

In recent years, the community has turned to improving randomized approaches. Manku et al. [32] present an algorithm that does not need advance knowledge of n, and showed its space requirements to be $O(\frac{1}{\epsilon}log^2(\frac{1}{\epsilon}))$. However, they must give up the deterministic guarantee on accuracy. Instead, they provide only a probabilistic guarantee that the quantile estimates are within the desired precision. Agarwal et al. [3] presented a mergeable algorithm with a space complexity of $O(\frac{1}{\epsilon}log^{1.5}(\frac{1}{\epsilon}))$. Karnin, Lang, and Liberty (KLL) [29] presented the KLL sketch, a randomized algorithm based on techniques from GK and the Quantiles sketch proposed in [3]. KLL is an asymptotically optimal non-mergeable sketch that solves the problem using $O(\frac{1}{\epsilon}\log\log(\frac{1}{\delta}))$ space. Karnin et al. also presented a mergeable KLL sketch with $O(\frac{1}{\epsilon}\log^2\log(\frac{1}{\delta}))$ space bounds.

Table 1.1: Comparison between different Quantiles sketches.

Algorithm	Space	Randomization	Mergeable
GK [22]	$O(\frac{1}{\epsilon}\log(\epsilon n))$	Deterministic	No
Q-digest [38]	$O(\frac{1}{\epsilon}\log(u))$	Deterministic	Yes
MRL [31]	$O(\frac{1}{\epsilon}\log^2(\epsilon n))$	Deterministic	Yes
MRL99 [32]	$O(\frac{1}{\epsilon}\log^2(\frac{1}{\epsilon})$	Randomized	Yes
KLL [29]	$O(\frac{1}{\epsilon}\log\log(\frac{1}{\delta})$	Randomized	No
Mergeable KLL [29]	$O(\frac{1}{\epsilon}\log^2\log(\frac{1}{\delta})$	Randomized	Yes
Quantiles Sketch [3]	$O(\frac{1}{\epsilon}\log^{1.5}(\frac{1}{\epsilon})$	Randomized	Yes

The mergeable Quantiles sketch proposed by Agarwal et al. [3] is very popular and forms a basis for many works that followed [29, 42]. The sequential solution proposed in [3] is used by Apache DataSketches [5], and our concurrent sketch is based on it. This Quantiles sketch is of sublinear-size and his estimates are *Probably Approximately Correct (PAC)*, providing an approximation within some error ϵn with a failure probability bounded by some parameter δ.

In the context of sequential processing, the classic literature on sketches has focused on inducing a small error while using a small memory footprint: The sketch is built by a single thread, and queries are served only 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 [37, 40]. Of these, the only solution

suitable for quantiles that we are aware of is the Fast Concurrent Data Sketches (FCDS) framework proposed by Rinberg et al. [37]. FCDS presents a generic algorithm for parallelizing data sketches efficiently and allowing them to be queried in real time. In general, FCDS is fast and achieves high scalability while keeping the estimation error small. The architecture of FCDS is based on local buffering of updates and constantly propagating results to a shared memory. Update threads ingest updates to local buffers. When a local buffer is full, its content is propagated to a shared global sketch. A single thread, called the propagator, propagates elements from all local buffers to the global sketch. Query threads access the global sketch. In the FCDS-based Quantile sketch, the local buffers are sequential Quantiles sketches. Each update thread updates its local sketch and when it is full, signals the propagator. As mentioned above, the propagation of the local sketches into the global sketch is made only by the propagator while other update threads may be idle. 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 in a high relaxation and 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 below). Our goal is to provide a scalable concurrent Quantiles sketch that retains a small error bound with reasonable query freshness.

In Chapter 2 we formally define the system model and the problem and also overview a popular sequential solution proposed by Agarwal et al. [3], which is used by Apache DataSketches [5], on which our concurrent sketch is based.

In Chapter 3, we present Quancurrent, our highly scalable concurrent Quantiles sketch. 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 shared sketch. Moreover, the shared sketch itself is organized in multiple levels, which may be propagated (and sorted) concurrently by multiple threads.

To allow queries to scale as well, Quancurrent serves them from a cached snapshot of the shared sketch. This architecture is illustrated in Figure 1.1. 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.

To lower synchronization overhead, 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*, alter the stream ingested by the data structure. Yet, in Chapter 4 we show that for a sufficiently large local buffer, the expected number of holes is less than 1 and because they are random, they do not change the sampled

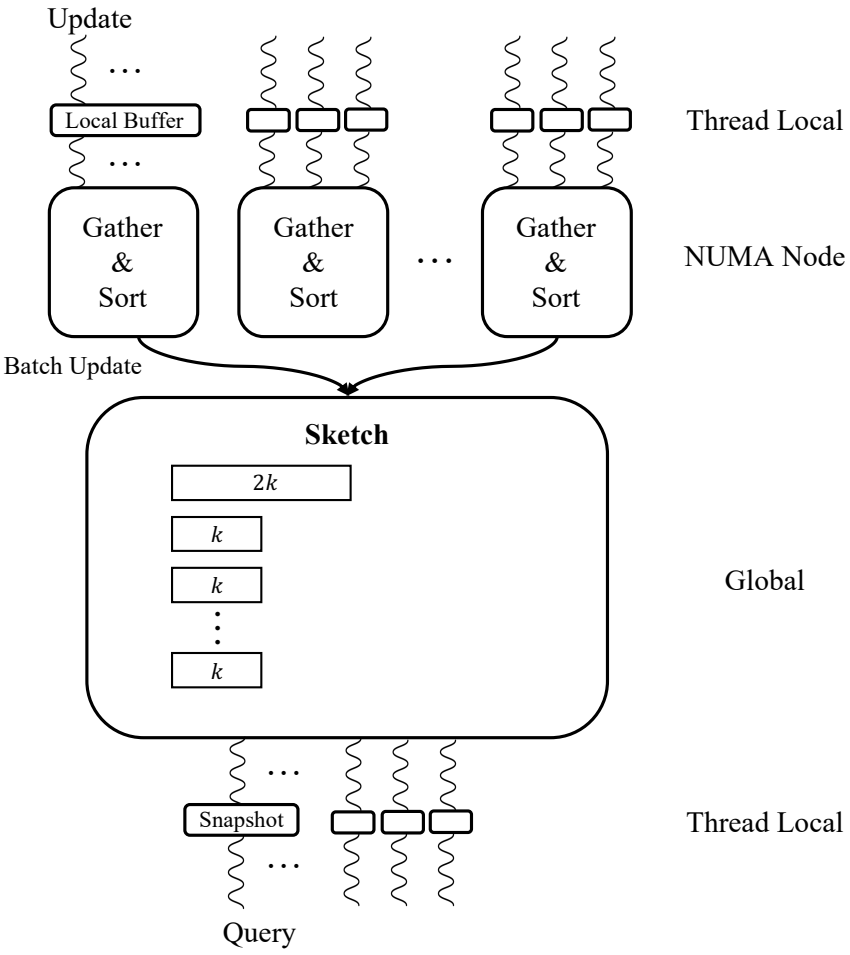


Figure 1.1: Quancurrent's architecture.

distribution.

Figure 1.2 presents quantiles estimated by Quancurrent on a stream of normally distributed random values compared to an exact, brute-force computation of the quantiles, and shows that the estimation is very accurate.

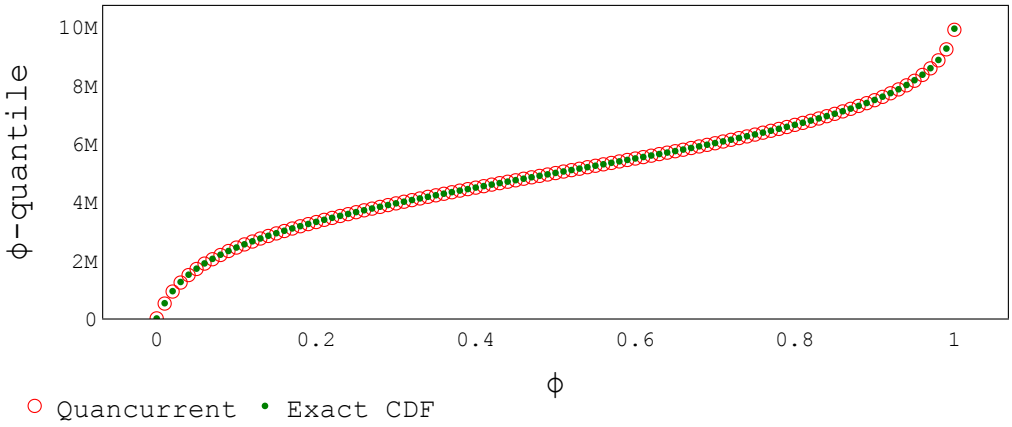


Figure 1.2: Quancurrent's ϕ-quantiles vs. exact quantiles (normal distribution, $k = 1024$, 32 update threads, $10M$ elements).

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

In Chapter 6 we present a formal correctness proof. Finally, Chapter 7 concludes our work and presents some open questions for future research.

1.1 Summary of Contributions

We present Quancurrent, a scalable Quantiles sketch that retains a small error bound with reasonable query freshness. The main technical challenges we address are:

1. High scalability. Quancurrent's throughput increases linearly with the number of available threads. High throughput can be achieved with much smaller buffers (hence better query freshness).

2. Accurate estimates (small error bound, fresh query). Quancurrent allows queries to occur concurrently with updates and achieves better query freshness than existing scalable solutions.

3. Eliminating sequential propagation bottleneck. Quancurrent allows more concurrency than previous solutions by utilizing multiple buffers and allowing multiple threads to

concurrently engage in merge-sorts, which are a sequential bottleneck in previous solutions.

4. Minimizing synchronization. We leverage the fact that sketches are approximate to begin with to dramatically reduce the synchronization overhead.

5. Correctness semantics with guaranteed error bounds. Quancurrent query's snapshot is strongly linearizable with respect to an r-relaxed sequential Quantiles sketch with $r = 4kS + (N - S)b$.

Chapter 2

Background

2.1 Preliminaries: Model and Correctness

We consider a shared memory model, where a finite number of threads execute *operations* on shared *objects*. An operation consists of an *invocation* and a matching *response*. A *history* H is a finite sequence of operation invocation and response steps. A history H defines a partial order $\prec_H$ on operations: Given operations *op* and *op*′, *op* $\prec_H$ *op*′ if and only if $response(op)$ precedes $invocation(op')$ in H. Two operations that do not precede each other are *concurrent*. In a *sequential history*, there are no concurrent operations. An object is specified using a *sequential specification* $\mathcal{H}$, which is the set of its allowed sequential histories. An operation *op* is *complete* in a history H if both $invocation(op)$ and its matching $response(op)$ are in H. A *linearization* of a concurrent history H, is a sequential history H' such that: (1) $H' \in \mathcal{H}$, (2) H' contains all completed operations and possibly additional non-complete ones, after adding matching responses, and, (3) $\prec_{H'}$ extends $\prec_H$. A correctness condition for randomized algorithms *strong linearizability* [21], defined as follows:

Definition 2.1.1 (strong linearizability)**.** A function f mapping executions to histories is *prefix-preserving* if for any two executions σ, σ', where σ is a prefix of σ', $f(\sigma)$ is a prefix of $f(\sigma')$. An object A is *strongly linearizable* if there is a prefix-preserving function f that maps every history H of A to a linearization of H.

Our algorithm is randomized and we consider a *weak adversary* that determines the scheduling without observing the coin-flips.

We adopt a flavor of *relaxed semantics*, as defined in [25]:

Definition 2.1.2 (r-relaxation)**.** A sequential history H is an *r-relaxation* of a sequential history H' if H is comprised of all but at most r of the invocations in H' and their responses, and each invocation in H is preceded by all but at most r of the invocations that precede the same invocation in H'. The r-relaxation of a sequential specification $\mathcal{H}$ is the set of histories that have r-relaxations in $\mathcal{H}$:

$$H^r \triangleq \left\{ H' \mid \exists H \in \mathcal{H} : H \text{ is an } r\text{-relaxation of } H' \right\}.$$

2.2 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 $R(A, x)$. A *quantile* is a value that is associated with a particular *rank*. For any $0 \leq \phi \leq 1$, the ϕ *quantile* of A is an element x such that $R(A, x) = \lfloor \phi n \rfloor$.

A Quantiles sketch's Application Programming Interface (API) is as follows:

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

A PAC Quantiles sketch with parameters ϵ, δ returns element x for query(ϕ) 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 ϕ quantile in a subset of the stream processed so far including all but at most r stream elements [25, 37].

2.3 Sequential Implementation

The Quantiles sketch proposed by Agarwal et al. [3] 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 (ϵ, δ). 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 2.1 depicts the processing of $4k$ elements.

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 ϕ 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 ϕ quantile is an element x_j, such that $W(x_j) \leq \lfloor \phi n \rfloor$ and $W(x_{j+1}) > \lfloor \phi n \rfloor$.

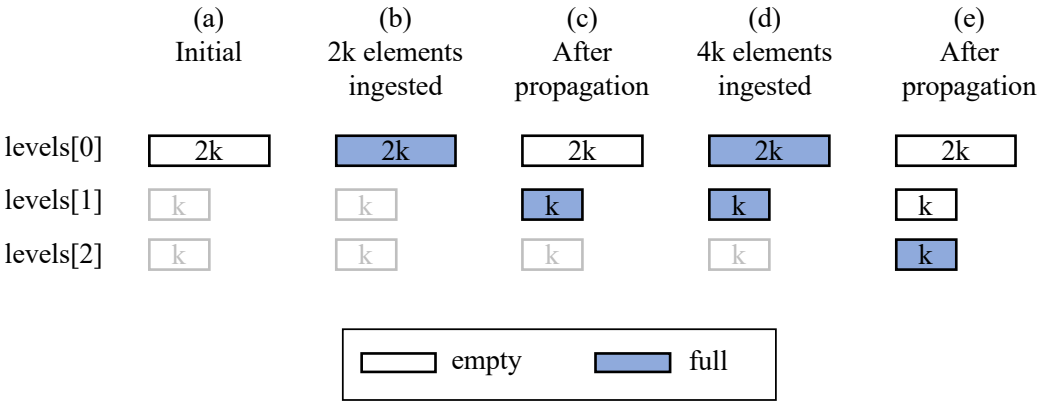


Figure 2.1: Quantiles sketch structure and propagation.

Chapter 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.

In Section 3.1, we present the memory model and the data structures used by Quancurrent. Section 3.2 presents the update operation, and Section 3.3 presents the query implementation.

3.1 Memory Model and Data Structures

3.1.1 Model

We consider a shared memory model that provides synchronization variables (atomics) and atomic operations to guarantee sequential consistency as in C++ [6]: 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)* [17] and *compare-and-swap (CAS)* [8].

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 a 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 [24, 23].

3.1.2 Data Structures

Quancurrent's data structures are described in Algorithm 3.1 and depicted in Figure 3.1. Similarly to the sequential Quantiles sketch, Quancurrent is organized as a hierarchy of arrays called *levels*. The maximum number of levels is MAX_LEVEL. 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 3.1a.

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 S is the number of NUMA nodes.

Algorithm 3.1 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:
7:  Shared objects:
8:      tritmap ← 0
9:      levels[MAX_LEVEL]
10:
11: NUMA-local objects:          ▷ shared among threads on the same node
12:     G&SBuffer[2][2k]
13:     index[2] ← {0, 0}
14:
15: Thread local objects:
16:     localBuf[b]
17:     myTrit                                     ▷ used by query
18:     snapshot                                   ▷ used by query
```

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.

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 3.2. 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 27). If the index does not overflow, the thread copies its local buffer to the reserved slots (Line 29). 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 31-32). As each update thread sorts its local buffer before moving it to the G&SBuffer, When full, the G&SBuffer consists of $\frac{2k}{b}$ sorted arrays (referred to as *regions*), each of size b. As such, the owner performs $\frac{2k}{b}$*-way merge* to sort the shared buffer.

Note that the local buffer is not atomically moved into the shared buffer (Line 29 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 both of these occurrences *holes* and analyze their implications in Section 4.1. These holes may cause some regions in the G&SBuffer to be not sorted when the owner is about to create a sorted local copy of the shared buffer. Thus, the owner thread creates a local copy of the shared buffer and then performs a single pass to make sure each region is monotonic and, if not, sorts it. Lastly, the owner uses $\frac{2k}{b}$-way merge to sort the full buffer.

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 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 3.3, and an example is depicted in Figure 3.1b.

In the beginning of the third stage, level 0 points to a new sorted copy of a *G&SBuffer*

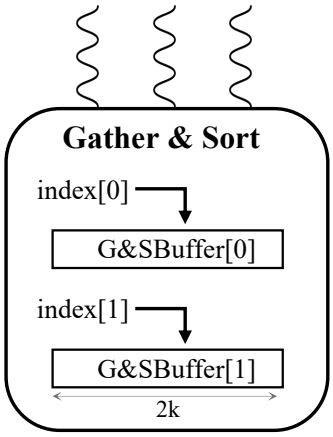
Algorithm 3.2 Stage 1: gather and sort

```
19: procedure UPDATE(x)
20:     add x to localBuf                                   ▷ thread-local
21:     if ¬localBuf.full() then
22:         return
23:     end if
24:     sort localBuf
25:     i ← 0
26:     while true do                                       ▷ insert to Gather&Sort unit
27:         idx ← index[i].F&A(b)
28:         if idx < 2k then
29:             move localBuf to G&SBuffer[i][idx, ..., idx + b]
30:             if idx + b = 2k then                        ▷ owner, filled buffer
31:                 myCopy ← sorted copy of G&SBuffer[i]
32:                 batchUpdate(i,myCopy)
33:             end if
34:             return
35:         end if
36:         i ← ¬i
37:     end while
38: end procedure
```

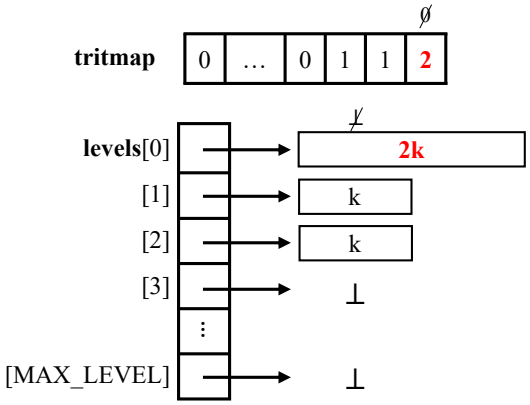
Algorithm 3.3 Stage 2: batch update

```
39: procedure BATCHUPDATE(i,base_copy)
40:     while ¬DCAS(levels[0]: ⊥ → base_copy, tritmap[0]: 0 → 2 do
41:         index[i] ← 0
42:         propagate(0)
43:     end while
44: end procedure
```

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 3.4. 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 49). 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 51). We then (in Line 52) continuously try, using DCAS, to update *levels*[l+1] to point to the merged array and atomically update *tritmap* such that *tritmap*[l] $\leftarrow 0$, reflecting level l is available, and *tritmap*[l+1] $\leftarrow 2$, reflecting that level l+1 contains $2k$ elements. After a successful DCAS, we clear level l (set it to $\perp$) and proceed to propagate the next level (Line 55). If $tritmap[l+1]$ is 0, then level $l+1$ is empty. We use DCAS (Line 57) to update $levels[l+1]$ to point to the sampled elements and atomically update *tritmap* so that *tritmap*[l] becomes 0,



(a) *Gather&Sort* object.



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

Figure 3.1: Quancurrent's data structures.

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 3.2 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 query algorithm is given in Section 3.3.1 and the snapshot's correctness is proven in Section 3.3.2.

Algorithm 3.4 Stage 3: Propagation of level l

```
45: procedure PROPAGATE(l)
46:     if l ≥ MAX_LEVEL then
47:         return
48:     end if
49:     newLevel ← sampleOddOrEven(levels[l])    ▷ choose odd or even indexed elements randomly
50:     if tritmap[l+1] = 1 then                 ▷ next level is full
51:         newLevel ← merge(newLevel, levels[l+1])
52:         while ¬DCAS(levels[l+1]: levels[l+1] → newLevel, tritmap[l, l+1]: [2,1] → [0,2]) do {}
53:         end while
54:         levels[l] ← ⊥                        ▷ clear level
55:         return propagate(l+1)
56:     end if
57:     while ¬DCAS(levels[l+1]: ⊥ → newLevel, tritmap[l, l+1]: [2,0] → [0,1]) do {}
58:     end while
59:     levels[l] ← ⊥                            ▷ clear level
60: end procedure
```

3.3.1 Query Algorithm

The pseudo-code is presented in Algorithm 3.6. 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 isn't 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 ρ, 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 75-76). Otherwise, a new snapshot is taken and cached.

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. The set of levels read between the two tritmap's reads are saved in $snapLevels$. 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 3.2, let's assume a query thread T_q reads $tm1 = 00202$, then reads the levels from $levels[0]$ to $levels[4]$ as depicted in Figure 3.2 (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 3.5. 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 87). Back to our

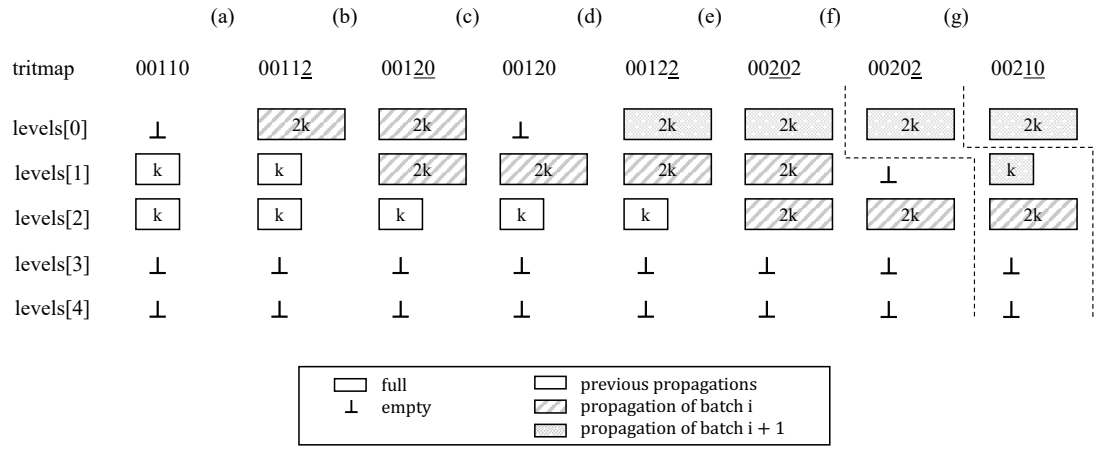


Figure 3.2: Quancurrent's propagation.

(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 $[1,0]$.

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 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 89).

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 following section.

3.3.2 Query's Snapshot Correctness

Let σ be an execution of Quancurrent and let q be a query operation executed by a query thread T in σ. Let $tm2$ be a response of the last read operation of the variable tritmap during q, denoted op_2, and let $tm1$ be the response of the penultimate read operation of tritmap,

Algorithm 3.5 Tritmap

```
61: procedure STREAMSIZE( )
62:     curr_stream ← 0
63:     for i ← 0, ..., MAX_LEVEL do
64:         weight ← 2^i
65:         if tritmap[i] = 1 then
66:             curr_stream ← curr_stream + weight·k
67:         else if tritmap[i] = 2 then
68:             curr_stream ← curr_stream + weight·2k
69:         end if
70:     end for
71:     return curr_stream
72: end procedure
```

denoted op_1, both executed by T between the invocation of the query q, $invocation(q)$, and the response of the same query, $response(q)$. Note that op_1 precedes op_2. Let A be the stream represented by the value of $tm1$. As tritmap represents the state of each level in Quancurrent, at the point of $tm1$, all the elements in Quancurrent summarize the stream A. We denote by $|A|$ the size of the stream. Let $snapLevels$ be the set of all levels read by T between op_1 and op_2, i.e., $snapLevels[i]$ points to the response of the read of level i between op_1 and op_2. We will prove that the constructed *snapshot* in Algorithm 3.6, Lines 83-94, summarizes the same stream A. By definition, the state of $snapshot$ is represented by the variable $myTrit$.

First, we show that $tm1$ and $tm2$ represent the same stream, i.e. A.

Lemma 3.3.1. *Let $tm1$ and $tm2$ be responses of two read operations of $tritmap$. If $tm1$ and $tm2$ represent streams with equal sizes then $tm1$ and $tm2$ represent the same stream.*

Proof. As described in Section 3.1.2, the i'th trit of the variable tritmap represents the state of Quancurrent's i'th level. Using Algorithm 3.5, we can calculate the size of the stream represented by a certain value of tritmap. The variable tritmap is atomically updated by DCAS operations such that the size of the stream represented by it is monotonic increasing. let tm_b be a value of the tritmap before an atomic DCAS, and let tm_a be the value of the tritmap after this DCAS. In case of DCAS failure, the value of tritmap has not changed. Thus, tm_b and tm_a represent the same stream. In case the DCAS succeeds, the value of the tritmap is updated. Using Algorithm 3.5, we show that the size of the stream represented is monotonic increasing:

- $tritmap[0]$: $[0] \rightarrow [2]$ (Algorithm 3.3, Line 40) –

$$stream\_size(tm_a) - stream\_size(tm_b) = (2^0 \cdot 2k) - (2^0 \cdot 0) = 2k$$

- $tritmap[i, i+1]$: $[2, 1] \rightarrow [0, 2]$ (Algorithm 3.4, Line 52) –

$$stream\_size(tm_a) - stream\_size(tm_b) = (2^i \cdot 2k + 2^{i+1} \cdot k) - (2^{i+1} \cdot 2k) = 0$$

Algorithm 3.6 Query

```
73: procedure QUERY(φ)
74:     tm1 ← tritmap
75:     if tm1.streamSize() / myTrit.streamSize() ≤ ρ then
76:         return snapshot.query(φ)
77:     end if
78:     repeat
79:         tm1 ← tritmap
80:         snapLevels ← read levels 0 to MAX_LEVEL
81:         tm2 ← tritmap
82:     until tm1.streamSize() = tm2.streamSize()
83:     myTrit ← 0
84:     snapshot ← empty snapshot
85:     for i ← MAX_LEVEL, ..., 0 do
86:         weight ← 2^i
87:         if snapLevels[i].size()·weight+ myTrit.streamSize()≤tm1.streamSize() then
88:             add snapLevels[i] to snapshot
89:             myTrit[i] ← snapLevels[i].size()/k
90:             if myTrit.streamSize()=tm1.streamSize() then
91:                 break
92:             end if
93:         end if
94:     end for
95:     return snapshot.query(φ)
96: end procedure
```

- $tritmap[i, i+1]$: $[2, 0] \rightarrow [0, 1]$ (Algorithm 3.4, Line 57) –

$$stream\_size(tm_a) - stream\_size(tm_b) = (2^i \cdot 2k) - (2^{i+1} \cdot k) = 0$$

Thus, the tritmap is updated such that the size of the stream represented by it is monotonic increasing. ■

Second, we show that $snapLevels$ contains all sampled elements summarizing the stream A in the sketch.

Lemma 3.3.2. *The set of levels, $snapLevels$, read between the two tritmap's reads, op_1 and op_2, contains all the elements contained in Quancurrent at the point of $tm1$.*

Proof. Let x be an element in level j immediately after $op1$ response. If x exists in level j during the read of sketch's levels in Algorithm 3.6 Line 80, then we are done. Otherwise, a propagation occurred in between the reads such that level j was merged with the next level and cleared. During this merge, all level j's elements, including x, were sampled and propagated to level $j + 1$. If x exists in level $j + 1$ in the set snapLevels then we are done, if not, we apply the above argument again. This continues up to MAX_LEVEL and therefore snapLevels contains the element x. ■

The following refers to the construction of the subset $snapshot$ from level MAX_LEVEL to 0:

Lemma 3.3.3. *While iterating $snapLevels[]$ from level MAX_LEVEL to 0 in Algorithm 3.6, Lines 85-94, if level j ($snapLevels[j]$) contains an element that is in $snapshot$, then all elements in level j are also in this $snapshot$.*

Proof. During a call to the procedure *propagate* with level j, the elements in that level are sampled and merged with level $j+1$, resulting in level $j+1$ representing also the elements of (former) level j. Let x be an element represented by level j in the set $snapLevels$. x is also represented by level i such that level i is in the current $snapshot$ and $i > j$. Considering the process of propagation from level j to level i, it follows from the above that level i also represents all the elements in level j. ■

As described in Section 2.3, each element in level j represents 2^j elements from the processed stream.

Lemma 3.3.4. *While iterating $snapLevels[]$ from level MAX_LEVEL to 0 in Algorithm 3.6, Lines 85-94, if level j ($snapLevels[j]$) contains an element that is in $snapshot$, then this element will not be inserted to $snapshot$ in the following iterations of lower levels, i.e., an element is inserted to $snapshot$ at most once.*

Proof. We prove that if level j represents new elements that are not represented by $snapshot$, the size of the sub-stream left to represent is at least the size of the representation of level j. We show this by contradiction. Assume by contradiction that the size of the sub-stream left to represent is smaller than the size of the representation of level j such that $|snapshot| \cup |level[j]| > |A|$. It follows from Lemma 3.3.2 that level j contains at least one duplicated element (an element already represented by level i in $snapshot$ such that $i > j$). From Lemma 3.3.3 all level j's elements are duplicated and already being represented by the current $snapshot$. Contradiction.

Now, we prove that if level j represents elements already represented by the current snapshot, the sub-stream left to represent is smaller than the representation of level j. Assume that an element x is represented by the current $snapshot$. Let level j be the second level to be added to $snapshot$ that also represents x (the first time to duplicate the representation of x). Note, element x has two representations in the current snapshot. From Lemma 3.3.3, all elements in level j are already represented by this snapshot. If level j contains $2k$ elements, the size of the sub-stream represented by level j is $2k \cdot 2^j$, and the size of the sub-stream left to represent is at most represented by levels 0 to $j-1$, meaning the size is at most $2k(1+2+\cdots+2^{j-1}) = 2k(2^j-1)$ which is smaller than the size of sub-stream represented by level j. If level j contains k elements and is already represented by level i, $i > j$, in $snapshot$, level $j-1$ must have propagated level j deeper, i.e. level $j-1$ is also represented by level i, therefore, also already represented by $snapshot$. Level j represents $k \cdot 2^j$. The size of the sub-stream left to represent is at most represented by levels 0 to $j-2$, meaning the size

is at most $2k(1+2+\cdots+2^{j-2}) = k(2^{j-1}-2)$, which is smaller than the size of sub-stream represented by level j. ■

Lemma 3.3.5. *The snapshot constructed in Algorithm 3.6, Lines 83-94, summarizes the same stream, A, as represented by the second tritmap read, $tm2$, in Algorithm 3.6, Line 81 (in the last iteration).*

Proof. From Lemma 3.3.4 $|snapshot| = |A|$ and every element is represented at most once in the constructed *snapshot*. ■

Chapter 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 a thread's local buffer to a shared G&SBuffer non-atomically, holes may occur when the owner thread reads older elements that were written to the shared buffer in a previous batch. The missed (delayed) writes may later overwrite 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* [4], 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 $\frac{1}{N}$, where N is the number of threads. Note that holes are random and the duplicate/missing elements are drawn from the stream's 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 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 G&SBuffer's owner, T_O, is $T_{\frac{2k}{b}}$. To initiate a batch update, T_O creates a local copy of his 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 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 hasn't 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 (a total of $b+(j-1)b+i+1$ steps), before T_j takes $i+1$ steps. The probability that T_O reads a hole in region j for the first time, in slot $i+1$ is:

$$\begin{aligned}\pi_{i,j} \triangleq P[\text{hole in slot } i+1 \mid \text{no hole in slots } 1\ldots i] \\ \cdot P[\text{no hole in slots } 1\ldots I].\end{aligned} \tag{4.1}$$

For a hole to be read in slot $i+1$ of region j (not necessarily for the first time in this region), 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,

$$\pi_{i,j} \leq \left(\frac{1}{2}\right)^{jb+2i+1} \binom{jb+2i}{i}. \tag{4.2}$$

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:

$$p_j \leq \sum_{i=0}^{b-1} \pi_{i,j}. \tag{4.3}$$

In the supplementary material (Claim A.3) we prove that the above upper bound of $\pi_{i,j}$ (Equation 4.2) is monotonically increasing for $i \in 0,1,\ldots,b-1$ and for $j \in \mathbb{N}$. Therefore,

$$\pi_{i,j} \leq \left(\frac{1}{2}\right)^{jb+2b-1} \binom{jb+2b-2}{b-1}. \tag{4.4}$$

Using Equation 4.3,

$$p_j \leq \sum_{i=0}^{b-1} \left(\frac{1}{2}\right)^{jb+2b-1} \binom{jb+2b-2}{b-1} \tag{4.5}$$

$$\leq b \cdot \left(\frac{1}{2}\right)^{jb+2b-1} \binom{jb+2b-2}{b-1} \tag{4.6}$$

If T_j has completed any number of 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:

$$E[H_j] = P(H_j = 0) \cdot 0 + P(H_j = 1) \cdot 1 + \cdots + P(H_j = b) \cdot b. \tag{4.7}$$

T_O can read at most b holes, therefore,

$$E[H_j] < b \cdot (P(H_j = 1) + \cdots + P(H_j = b)) \tag{4.8}$$

$$= b \cdot P(H_j \geq 1) < b \cdot p_j. \tag{4.9}$$

Next we show that $E[H_1] \leq 1.4$ for all b.

Lemma 4.1.1. *$E[H_1] \leq 1.4$ for all $b \in \mathbb{N}$.*

Proof. Denote

$$g(b) \triangleq b^2 \cdot \left(\frac{1}{2}\right)^{3b-1} \binom{3b-2}{b-1} \tag{4.10}$$

From Equation 4.9, $E[H_1]$ is bounded by

$$E[H_1] \leq b^2 \cdot \left(\frac{1}{2}\right)^{3b-1} \binom{3b-2}{b-1} = g(b) \tag{4.11}$$

First, we show that $g(b)$ is monotonically decreasing for $b \geq 12$.

$$\frac{g(b+1)}{g(b)} = \frac{(b+1)^2 \cdot \left(\frac{1}{2}\right)^{3b+2} \binom{3b+1}{b}}{b^2 \cdot \left(\frac{1}{2}\right)^{3b-1} \binom{3b-2}{b-1}} \tag{4.12}$$

$$= \left(\frac{1}{2}\right)^3 \cdot \left(\frac{3}{2}\right) \frac{(b+1)^2}{b^2} \cdot \frac{3b-1}{b} \cdot \frac{3b+1}{2b+1} \tag{4.13}$$

In the Supplementary Material (Claims A.4-A.6), we show that for $b > 12$

$$\frac{g(b+1)}{g(b)} = \left(\frac{1}{2}\right)^3 \cdot \left(\frac{3}{2}\right) \underbrace{\frac{(b+1)^2}{b^2}}_{\leq \left(\frac{13}{12}\right)^2} \cdot \underbrace{\frac{3b-1}{b}}_{<3} \cdot \underbrace{\frac{3b+1}{2b+1}}_{<\frac{3}{2}} < 1 \tag{4.14}$$

Therefore, $g(b)$ is monotonically decreasing for $b \geq 12$. Lastly,

$$\max_{1 \leq b \leq 12} \{f(b)\} = f(9) = 1.305 < 1.4. \tag{4.15}$$

That is, for all $b \in \mathbb{N}$,

$$g(b) < 1.4. \tag{4.16}$$

From Equation 4.11 and Equation 4.16,

$$\forall b \in \mathbb{N},\ E[H_1] \leq 1.4. \qquad \blacksquare$$

Lemma 4.1.2. *If $E[H_j] \leq \alpha_j$, then $E[H_{j+1}] \leq \frac{1}{2}\alpha_j$ for all $b \in \mathbb{N}$ and $j \geq 1, j \in \mathbb{N}$.*

Proof. From Equation 4.9,

$$E[H_j] \leq b^2 \left(\frac{1}{2}\right)^{bj+2b-1} \cdot \binom{jb+2b-2}{b-1}. \tag{4.17}$$

$$E[H_{j+1}] \leq b^2 \left(\frac{1}{2}\right)^{bj+3b-1} \cdot \binom{jb+3b-2}{b-1}. \tag{4.18}$$

Denote

$$\alpha_j \triangleq b^2 \left(\frac{1}{2}\right)^{bj+2b-1} \cdot \binom{jb+2b-2}{b-1}. \tag{4.19}$$

We show that $E[H_{j+1}] \leq \frac{1}{2}\alpha_j$.

$$E[H_{j+1}] \leq b^2 \left(\frac{1}{2}\right)^{bj+3b-1} \cdot \binom{jb+3b-2}{b-1} \tag{4.20}$$

$$= \frac{1}{2} \cdot b^2 \left(\frac{1}{2}\right)^{bj+2b-1} \cdot \left(\frac{1}{2}\right)^{b-1} \binom{jb+3b-2}{b-1} \tag{4.21}$$

Using Claim A.7

$$\leq \frac{1}{2} \cdot b^2 \left(\frac{1}{2}\right)^{bj+2b-1} \cdot \binom{jb+2b-2}{b-1} \tag{4.22}$$

$$= \frac{1}{2}\alpha_j. \tag{4.23}$$

Therefore, $E[H_{j+1}] \leq \frac{1}{2}\alpha_j$. $\blacksquare$

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

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

From Lemma 4.1.1 and Lemma 4.1.2,

$$E[H] = \sum_{j=1}^{\frac{2k}{b}} E[H_j] \tag{4.24}$$

$$\leq \sum_{j=1}^{\frac{2k}{b}} \left(\frac{1}{2}\right)^{j-1} \cdot E[H_1] \tag{4.25}$$

$$\leq \sum_{j=1}^{\infty} \left(\frac{1}{2}\right)^{j-1} \cdot E[H_1] \tag{4.26}$$

$$\leq \underbrace{\sum_{j=1}^{\infty} \left(\frac{1}{2}\right)^{j-1}}_{<2} \cdot 1.4 \tag{4.27}$$

$$\leq 2 \cdot 1.4 = 2.8 \tag{4.28}$$

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 the 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 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 might have locally buffered elements. Therefore, the buffering relaxation r is $4kS + (N - S)b$.

Rinberg et al. [37] show that for a query of a ϕ-quantile, an r-relaxation of a Quantiles sketch with parameters ϵ_c and δ_c, returns an element, x, such that

$$rank(x) \in [(\phi - \epsilon_r)n, (\phi + \epsilon_r)n]$$

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 ρ. 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 rank of the element returned by the cached snapshot is in the range:

$$[(\phi - \epsilon_r)\, n_{old}, (\phi + \epsilon_r)\, n_{old}] \tag{4.29}$$

As $n_{old} \leq n_{new}$ and $\epsilon' \geq 0$ then,

$$(\phi + \epsilon_r)\, n_{old} \leq (\phi + \epsilon_r)\, n_{new} \leq \left(\phi + \left(\epsilon' + \epsilon_r\right)\right) n_{new} \tag{4.30}$$

On the other hand, as $n_{old} \geq \frac{n_{new}}{\rho}$,

$$(\phi - \epsilon_r)\, n_{old} \geq (\phi - \epsilon_r)\, \frac{n_{new}}{\rho} = \qquad (\rho = 1 + \epsilon') \tag{4.31}$$

$$\left(\frac{\phi}{1+\epsilon'} - \frac{\epsilon_r}{1+\epsilon'}\right) n_{new} = \tag{4.32}$$

$$\left(\frac{\phi + \phi\epsilon' - \phi\epsilon'}{1+\epsilon'} - \frac{\epsilon_r}{1+\epsilon'}\right) n_{new} = \tag{4.33}$$

$$\left(\frac{\phi(1+\epsilon')}{1+\epsilon'} - \frac{\phi\epsilon'}{1+\epsilon'} - \frac{\epsilon_r}{1+\epsilon'}\right) n_{new} = \quad (0 \leq \frac{\phi}{1+\epsilon'} \leq \frac{1}{1+\epsilon'}) \tag{4.34}$$

$$\left(\phi - \frac{\epsilon'}{1+\epsilon'} - \frac{\epsilon_r}{1+\epsilon'}\right) n_{new} \geq \tag{4.35}$$

$$\left(\phi - \frac{1}{1+\epsilon'}\left(\epsilon' + \epsilon_r\right)\right) n_{new} \geq \qquad (\frac{1}{1+\epsilon'} \leq 1) \tag{4.36}$$

$$\left(\phi - \left(\epsilon' + \epsilon_r\right)\right) n_{new} \tag{4.37}$$

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

$$[(\phi - \epsilon)\, n, (\phi + \epsilon)\, n] \tag{4.38}$$

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

Chapter 5

Evaluation

In this chapter, we measure Quancurrent's throughput and accuracy of estimation. 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

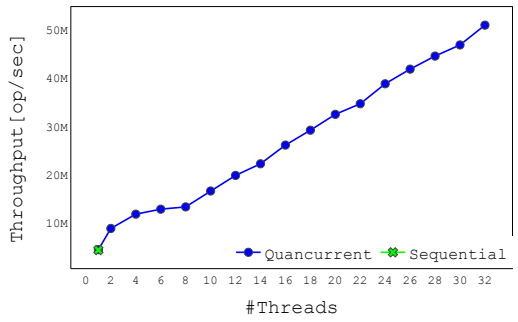
We implement Quancurrent in C++. Our memory management system is based on IBR [43], an interval-based approach to memory reclamation for concurrent data structures. The experiments were run on a NUMA system with four Intel Xeon E5-4650 processors, each with 8 cores, for a total of 32 threads (with hyper-threading 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.

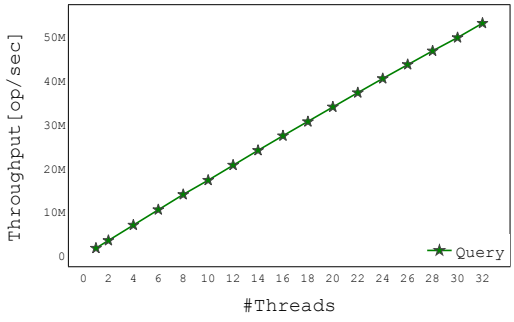
5.2 Throughput Scalability

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 5.1 shows Quancurrent's throughput in those workloads with $k = 4096$ and $b = 16$,

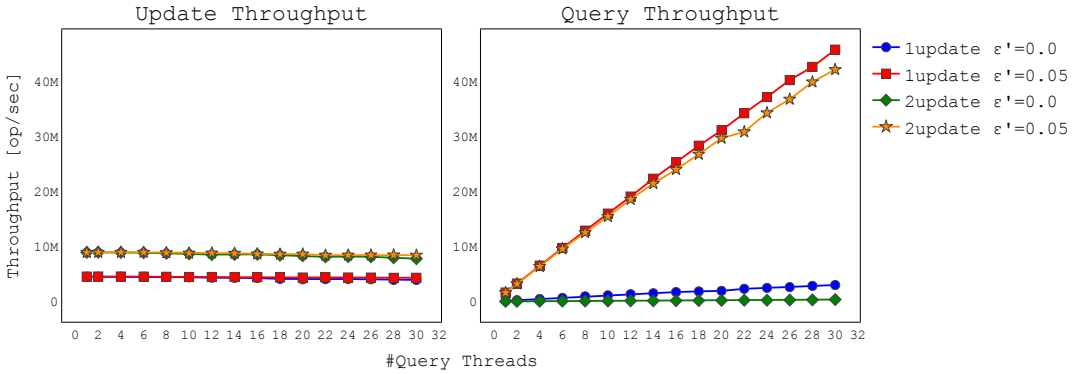
As shown in Figure 5.1a, Quancurrent's performance in the update-only workload with



(a) Update-only, $10M$ elements.



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



(c) One or two update threads, up to 32 query threads, $10M$ elements inserted after a pre-fill of $10M$ elements.

Figure 5.1: Quancurrent's throughput, $k = 4096$, $b = 16$.

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 5.1b 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 ρ is significant for performance - when $\rho = 1$ ($\epsilon' = 0$, no caching), a snapshot is reproduced on every query. Figure 5.1c 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 5.2a 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 5.2b 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 5.2c we vary ρ, 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 ρ has a positive impact on query throughput, as the cached snapshot can be queried more often. Figure 5.2c also shows the miss rate, which is the percentage of queries that need to re-construct the snapshot.

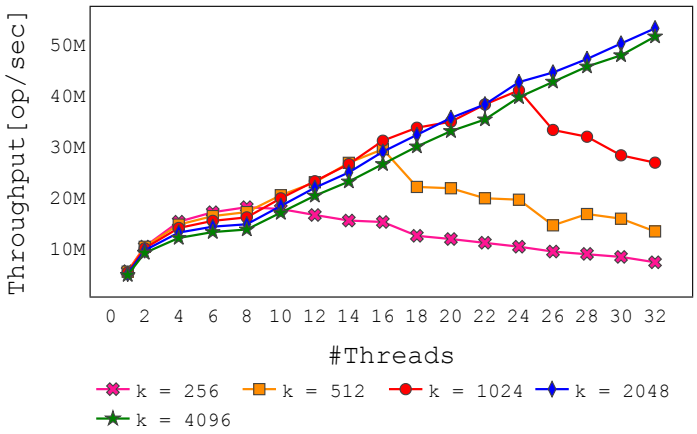
5.4 Accuracy

To measure the estimate accuracy, we consider a query invoked in a quiescent state where no updates occur concurrently with the query. Figure 5.3 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 improve with larger values of k as known from the literature on sequential sketches [3].

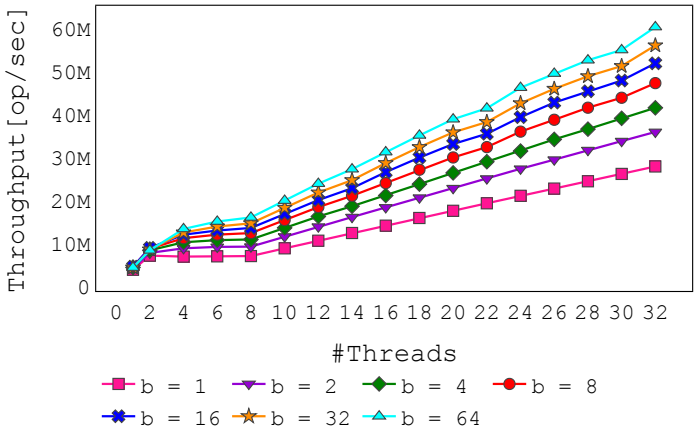
To illustrate the impact of k visually, Figure 5.4 compares the distribution measured by Quancurrent (red open-circles) to the exact (full information) stream distribution (green CDF filled-circles). In Figure 1.2 (in the introduction), we depict the accuracy of Quancurrent's estimate of a normal distribution with $k = 1024$. Figure 5.4b (left) shows that when we reduce k to 32, the approximation is less tight while for $k = 256$ (Figure 5.4b right) it is very accurate. We observe similar results for the uniform distribution in Figure 5.4a.

5.5 Comparison to State of the Art

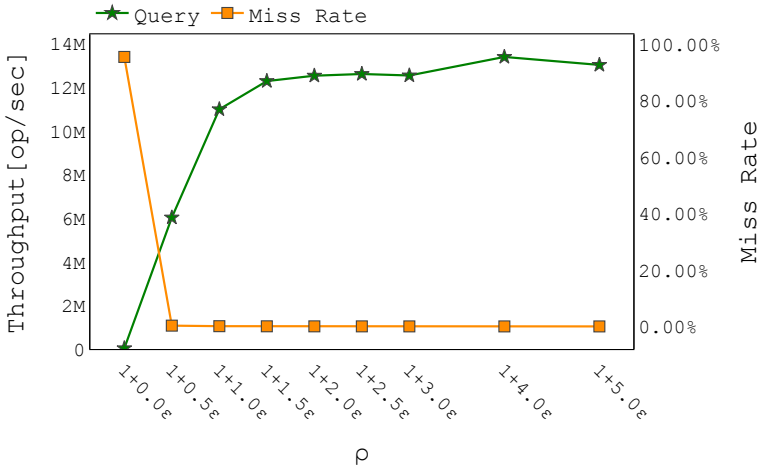
Finally, we compare Quancurrent against a concurrent Quantiles sketch implemented within the FCDS framework [37], the only previously suggested concurrent sketch we know that supports quantiles. Figure 5.5 (and Table 5.1) show the throughput results (log scale) for 8, 16, 24 and 32 threads and $k = 4096$. 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.



(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 5.2: Quancurrent's parameters impact.

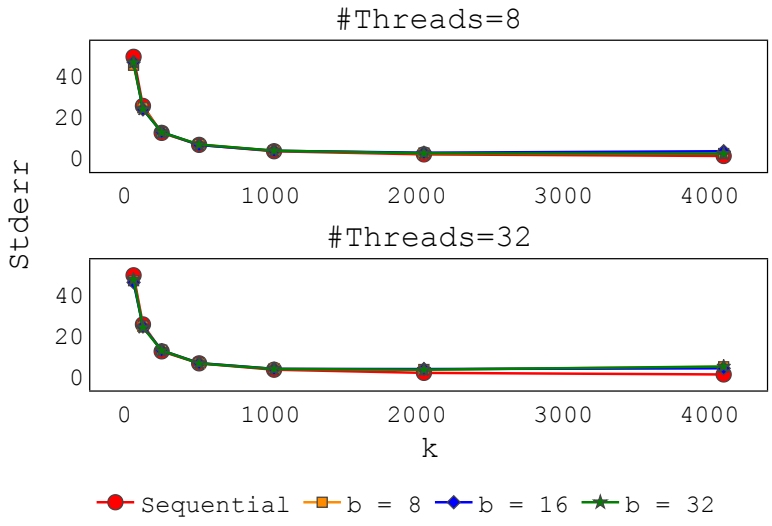
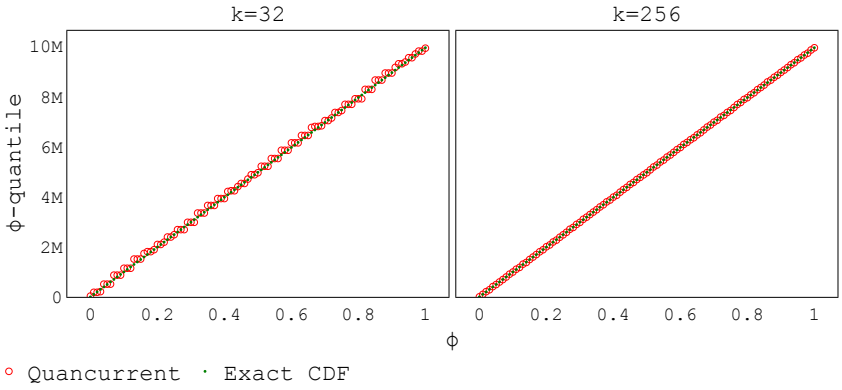
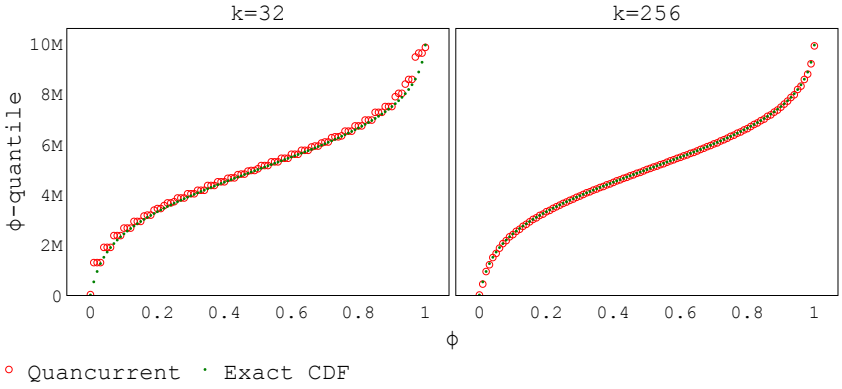


Figure 5.3: Standard error of estimation in quiescent state, keys = $1M$, runs = 1000.



(a) Uniform distribution.



(b) Normal distribution.

Figure 5.4: Quancurrent's ϕ-quantiles vs. exact quantiles, with 32 threads, $b = 16$, and a stream size of $10M$.

Recall that Quancurrent's relaxation is at most $r = 4kS + (N - S)b$. Thus:

$$r_{\text{FCDS}} = 2NB \tag{5.1}$$

$$r_{\text{Quancurrent}} = 4kS + (N - S)b \tag{5.2}$$

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

$$r_{\text{FCDS}} = r_{\text{Quancurrent}} \tag{5.3}$$

$$2NB = 4kS + (N - S)b \tag{5.4}$$

$$B = \frac{4kS + (N - S)b}{2N} \tag{5.5}$$

Given the sketch parameter k, the number of update threads N, the number of NUMA nodes S, and a series of Quancurrent' local buffer sizes b_i such that $b = \{2^i : i = 0, 1, \dots\}$, we calculate the corresponding B_i values (FCDS's local buffer size), using Equation 5.5.

The results are listed in Table 5.1. Note that the size of the local buffer b is bounded by the size of the *G&SBuffer* array, which is $2k$. Each row lists the parameters needed to get equal relaxation in both algorithms. We calculated the throughput for each algorithm and the results are shown in Figure 5.5. For clarity, some points with the same relaxation were colored the same in both lines. In addition, we specify the size of the buffer.

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 $131K$. Also, 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.

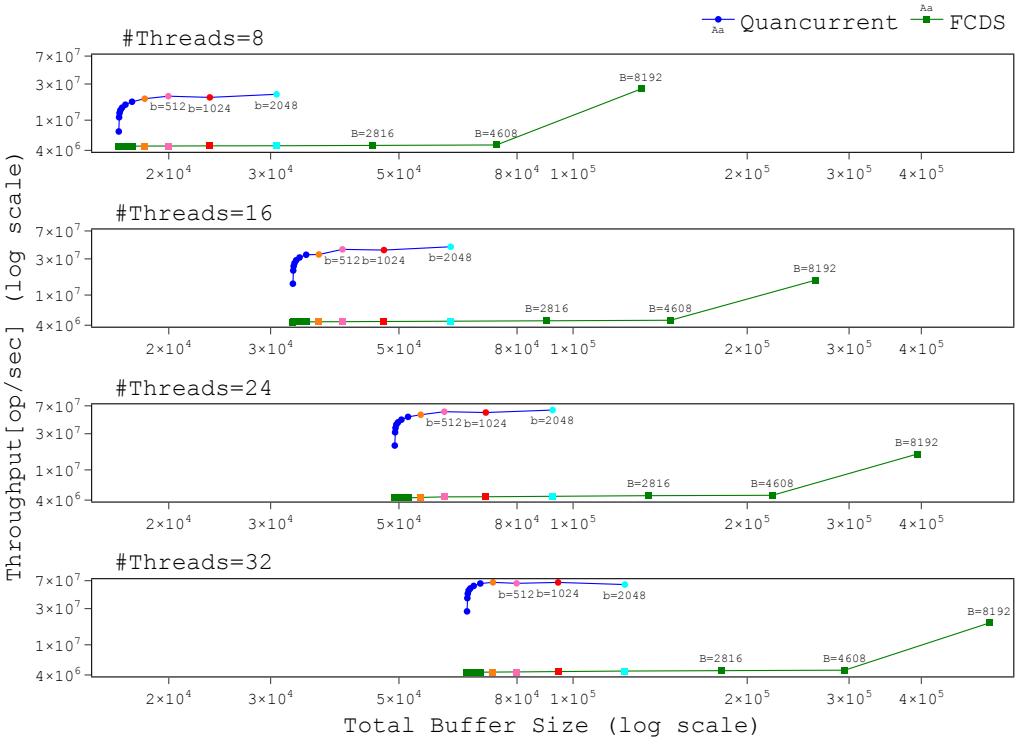


Figure 5.5: Quancurrent vs. FCDS, $k = 4096$.

Table 5.1: Quancurrent vs. FCDS, k = 4096, #keys = 10M

Quancurrent			FCDS		
N	b	Throughput$[\frac{op}{sec}]$	r	B	Throughput$[\frac{op}{sec}]$
8	1	7.1 M	16.4 k	1024	4.5 M
	4	10.9 M	16.4 k	1026	4.6 M
	8	12.4 M	16.4 k	1028	4.6 M
	16	13.5 M	16.5 k	1031	4.6 M
	32	14.6 M	16.6 k	1038	4.5 M
	64	16 M	16.8 k	1052	4.5 M
	128	17.5 M	17.3 k	1080	4.5 M
	256	19.2 M	18.2 k	1136	4.6 M
	512	20.7 M	20 k	1248	4.6 M
	1024	20 M	23.6 k	1472	4.6 M
	2048	22 M	30.7 k	1920	4.6 M
	4096	13 M	45.1 k	2816	4.7 M
	8192	7.13 M	73.7 k	4608	4.7 M
			131 k	8192	25.8 M
16	1	14.1 M	32.8 k	1024	4.4 M
	4	21.1 M	32.8 k	1026	4.4 M
	8	24 M	32.9 k	1028	4.4 M
	16	26.3 M	33 k	1031	4.5 M
	32	28.8 M	33.2 k	1038	4.4 M
	64	31.3 M	33.7 k	1052	4.4 M
	128	34 M	34.6 k	1080	4.4 M
	256	34.3 M	36.4 k	1136	4.4 M
	512	40 M	39.9 k	1248	4.5 M
	1024	39.2 M	47.1 k	1472	4.5 M
	2048	43.3 M	61.4 k	1920	4.6 M
	4096	25.3 M	90.1 k	2816	4.6 M
	8192	14 M	147 k	4608	4.7 M
			262 k	8192	15.7 M
24	1	21 M	49.2 k	1024	4.3 M
	4	31.4 M	49.2 k	1026	4.3 M
	8	35.8 M	49.3 k	1028	4.4 M
	16	39.6 M	49.5 k	1031	4.3 M
	32	42.4 M	49.8 k	1038	4.4 M
	64	46 M	50.5 k	1052	4.4 M
	128	50.1 M	51.8 k	1080	4.3 M
	256	53.3 M	54.5 k	1136	4.3 M
	512	58.6 M	59.9 k	1248	4.4 M
	1024	57.1 M	70.7 k	1472	4.4 M
	2048	61.5 M	92.2 k	1920	4.5 M
	4096	36.9 M	135 k	2816	4.6 M
	8192	20.8 M	221 k	4608	4.6 M
			393 k	8192	16.2 M
32	1	27.6 M	65.5 k	1024	4.3 M
	4	41.2 M	65.7 k	1026	4.3 M
	8	47.2 M	65.8 k	1028	4.3 M
	16	51.9 M	66 k	1031	4.3 M
	32	55.4 M	66.4 k	1038	4.3 M
	64	59.7 M	67.3 k	1052	4.4 M
	128	64.3 M	69.1 k	1080	4.4 M
	256	66.5 M	72.7 k	1136	4.4 M
	512	64.6 M	79.9 k	1248	4.4 M
	1024	66.4 M	94.2 k	1472	4.4 M
	2048	62.3 M	123 k	1920	4.5 M
	4096	48.1 M	180 k	2816	4.6 M
	8196	27.3 M	295 k	4608	4.6 M
			524 k	8192	19.4 M

Chapter 6

Correctness

In this section, we prove Quancurrent's correctness.

6.1 Preliminaries

Queries are answered from an array of ordered tuples summarizing the total stream processed so far, denoted as $samples$. Each tuple contains a summary point (i.e., a value from the sketch) and its associated weight. The samples array contains all the sketch's summary points and is sorted according to their values. Note that, only levels such that $tritmap[i] \in 1, 2$ are included.

As described in Section 3.2, the update operation is divided into 3 stages: (1) $gather\ and\ sort$ is the process of ingesting stream elements into a $Gather\&Sort$ unit. (2) $batch\ update$ is the process of copying $2k$ elements from one of the *G&SBuffer*s into Quancurrent's first level. (3) $propagate\ levels$ is the process of merging the base level up the sketch's levels until reaching an empty level.

6.1.1 Definitions

Rinberg et al. [37] defined the relation between a sequential history and a stream:

Definition 6.1.1. Given a finite sequential history H, $\mathcal{S}(H)$ is the stream $a_1, \ldots, a_n$ such that a_k is the argument of the k^{th} update in H.

The notion of *happens before* in a sequential history as defined in [37]:

Definition 6.1.2. Given a finite sequential history H and two method invocation M_1, M_2 in H, if M_1 precedes M_2 in H, we denote $M_1 \prec_H M_2$.

Definition 6.1.3 (Unprop updates)**.** Given a finite execution σ of Quancurrent, we denote by suffix(σ) as the suffix of σ starting at the last successful $batch\ update$ event, or the beginning of σ if no such event exists. We denote by $up\_suffix(\sigma)$ the sub-sequence of $H(suffix(\sigma))$ consisting of updates operations in the $Gather\&Sort$ units. We denote by $up\_suffix_i(\sigma)$ the

sub-sequence of $H(\mathit{suffix}(\sigma))$ consisting of updates operations in the local buffer of thread T_i.

Definition 6.1.4 (Updates Number)**.** We denote the number of updates in history H as $|H|$.

6.2 Algorithm Correctness Proof

Lemma 6.2.1. *Quancurrent is strongly linearizable with respect to r-relaxed sequential Quantiles sketch with $r = 4kS + (N - S)b$, where S is the number of NUMA nodes, k is the sketch summary size, b is the size of threads local buffer and N is the number of update threads.*

Proof. Quancurrent is an r-relaxed concurrent Quantiles sketch. The correctness condition for randomized algorithms under concurrency is strong linearizability [21]. Strong linearizability is defined with respect to the sequential specification. The sequential specification is defined with respect to deterministic objects. Therefore, we de-randomized Quancurrent by providing coin flips with every update. We denote by *SeqSketch* to be the sequential specification (i.e., the set of all sequential histories) of Quancurrent.

A relaxed consistency extends the sequential specification of an object to a larger set that contains sequential histories which are not legal but are at bounded "distance" from a legal sequential history [25, 2, 37]. We re-define Quancurrent sequential specification by relaxing it. Intuitively, we allow a query to "miss" a bounded number of updates that precede it. Quantiles sketch is order agnostic, thus re-ordering updates is also allowed. We denote by $SeqSketch^r$ the set of "relaxed" sequential histories.

Let σ be a concurrent execution of Quancurrent. We use two mappings, from concurrent executions to sequential histories, defined as follows. First, we define a mapping, l, from a concurrent execution to a serialization, by ordering operations according to the following linearization points:

- **Query** linearization point is the second *tritmap* read, $tm2$, such that it summarizes the same stream size as $tm1$ (Algorithm 3.6, Line 82).
- **Update** linearization point is the insertion of elements to threads' local buffers (Algorithm 3.2, Line 20).

Strong linearizability requires that the linearization of a prefix of a concurrent execution is a prefix of the linearization of the whole execution. By definition, $l(\sigma)$ is prefix-preserving. Note that $l(\sigma)$ is a serialization that does not necessarily meet the sequential specification.

Relaxed consistency extends the sequential specification of an object to include also relaxed histories. We define a second mapping, f, from a concurrent execution to a serialization, by ordering operations according to visibility points:

- **Query** visibility point is the query's linearization point.

- **Update** visibility point is the time after the update's invocation in σ in which the *G&SBuffer* that includes this update, is batched updated into level 0 of the global sketch. If there is no such time, the update does not have a visibility point, meaning, it is not included in the relaxed history, $f(\sigma)$.

To prove correctness we need to show that for every execution σ of Quancurrent: (1) $f(\sigma) \in SeqSketch$, meaning, the serialization according to visibility points is a "legal" sequential history of Quancurrent. and (2) $f(\sigma)$ is an r-relaxation of $l(\sigma)$ for $r = 4kS + (N - S)b$. That is, the sequential history $f(\sigma)$, comprised of all but at most r of the operations in $l(\sigma)$, and each invocation in $f(\sigma)$ is preceded by all but at most r of the invocations that precede the same invocation in $l(\sigma)$.

We show the first part.

Lemma 6.2.2. *Given a finite execution σ of Quancurrent, $f(\sigma)$ is in the sequential specification.*

Proof. First, we present and prove some invariants.

Invariant 1. *The Gather&Sort object summarises at most 4k elements.*

Proof. The Gather&Sort unit contains two buffers of $2k$ elements. Elements are ingested into threads' local buffers and moved to the G&SBuffer without being sampled. The desired summary is agnostic to the processing order, therefore S summarises history of $4k$ update operations and their responses. ■

Invariant 2. *The variable tritmap is a monotonic increasing integer.*

Proof. The variable tritmap is altered only in Line 40 of Algorithm 3.3, in Line 52 of Algorithm 3.4 and in Line 57 of Algorithm 3.4. By definition, it is only incremented. ■

Invariant 3. *The variable tritmap represents the sketch state:*

- *If $tritmap[i] = 0$, then $levels[i]$ is empty or is not contained in the sketch's samples array (needs to be ignored).*
- *If $tritmap[i] = 1$, then $levels[i]$ contains k points associated with a weight of 2^i.*
- *If $tritmap[i] = 2$, then $levels[i]$ contains $2k$ points associated with a weight of 2^i.*

Proof. The proof is by induction on the length of the levels array (or its current maximum depth).
Base: By definition, tritmap is initialized with 0 and updated during the batchUpdate procedure and the propagate procedure. After the first batch update, level 0 contains $2k$ elements and tritmap is increased by 2 such that tritmap[0]=2. When this first batch is merged with the next level, level 1 contains k elements and tritmap is increased by 1 such that tritmap[0]=0. On each propagation, we first perform a batch update of one of the G&SBuffer arrays to

level 0 and increase the tritmap by 2. Then we call propagate() starting with level 0. Level 0 is merged with the next level and the tritmap is incremented by 1. Therefore, after each batchUpdate, $tritmap[0] = 2$ and level 0 contains $2k$ elements and after each call to propagate(0) $tritmap[0] = 0$ and level 0 is not contained in the sketch's samples array. The following calls to propagate increase tritmap by 3^i for $i > 0$ and $tritmap[0] = 0$ until the end of the current propagation.
Inductive hypothesis: We assume the invariant holds for all levels i such that $i > 0$ and prove it holds for level $i + 1$. By definition, tritmap is updated during batchUpdate and propagate procedures. For $i > 0$, $tritmap$ is changed only if $tritmap[i] = 2$. By the inductive hypothesis, if $tritmap[i] = 2$, then $levels[i]$ contains $2k$ points associated with a weight of 2^i. If propagation has not yet reached level i+1, it is empty and $tritmap[i+1] = 0$ (by initialization). After a call to propagate(i), $levels[i + 1]$ contains k points associated with a weight of 2^{i+1} and tritmap satisfies $[b_{31}, \ldots, b_{i+2}, 0, 2, b_{i-1}, \ldots, b_0] + 3^i = [b_{31}, \ldots, b_{i+2}, 1, 0, b_{i-1}, \ldots, b_0]$ i.e $tritmap[i+1] = 1$. After the next call to propagate(i), level i+1 will contain $2k$ points associated with a weight of 2^{i+1} and tritmap will satisfy $[b_{31}, \ldots, b_{i+2}, 1, 2, b_{i-1}, \ldots, b_0] + 3^i = [b_{31}, \ldots, b_{i+2}, 2, 0, b_{i-1}, \ldots, b_0]$ i.e $tritmap[i + 1] = 2$. Note that each propagation starts from level 0 and stops when reaching an empty level j, the tritmap trit larger than j are not changed. ■

Invariant 4. *Given a finite execution σ of Quancurrent, it summarises $f(\sigma)$.*

Proof. The proof is by induction on the length of σ.
Base: The base is immediate. Quancurrent summarises the empty history.
Inductive hypothesis: We assume the invariant holds for σ', and prove it holds for $\sigma = \{\sigma', step\}$. We consider only steps that can alter the invariant, meaning steps that can change the sketch state.

- DCAS operation during the batchUpdate procedure, increasing tritmap by 2 and copying one of the G&SBuffer arrays into the first level of Quancurrent.

  By the inductive hypothesis, before the step, Quancurrent summarises $f(\sigma')$. If the DCAS fails, the sketch state has not changed. Else, $2k$ elements were copied to level 0 and the tritmap was increased by 2. From Invariant 1, a G&SBuffer array summarises a collection of $2k$ elements $\{a_1, \ldots, a_{2k}\}$. By copying, we sequentially ingest the stream $B = \{a_1, \ldots, a_{2k}\}$ to Quancurrent. Let $A = \mathcal{S}(f(\sigma'))$. By definition, Quancurrent summarises $A||B$. Therefore Quancurrent summarises $f(\sigma)$, preserving the invariant.

- DCAS operations during the propagate procedure, updating tritmap and merging level i with its following level.

  By the inductive hypothesis, before the step, Quancurrent summarises $f(\sigma')$. If the DCAS fails, the sketch state has not changed by the step. Else, we propagated level i into level $i + 1$. By definition, k points from level i were merged with level $i + 1$, with

the weight of each point scaled up by a factor 2. After the merge, tritmap[i]=0 therefore, level i was disabled from $samples[]$ and $2k$ points associated with a weight of 2^i are not included in the summary. The total weight of the sketch's elements was not changed. The sketch summarises the same stream, no new points were added and the stream size was not changed. Thus, the sketch's state presents (summarises) the same stream, that is, Quancurrent summarises $f(\sigma)$, preserving the invariant.

- Operations to clear level i, updating $levels[i] \leftarrow \perp$.

  By definition, $tritmap[i] = 0$. By Invariant 3, $levels[i]$ is empty or ignored and not included in $samples[]$. Thus, operations to clear levels do not affect the stream processed by the sketch thus far. Quancurrent summarises $f(\sigma)$, preserving the invariant. ■

Lemma 6.2.3 (Query Correctness)**.** *Given a finite execution of Quancurrent σ, let Q be a query operation that returns in σ. Let v be the visibility point of Q, and let σ' be the prefix of σ until the point v. The query Q returns a value equal to the value returned by a query operation of the sequential Quantiles sketch after processing the stream $\mathcal{S}\,(f(\sigma'))$.*

Proof. Let σ, Q, v and σ' be as defined in the Lemma, and let $A = \mathcal{S}(f(\sigma'))$. By definition, the visibility point of a query is when the second tritmap read returns a value representing the same stream size as the previous tritmap read. As proved in Lemma 3.3.5, the collected snapshot summarizes the same stream as Quancurrent at the visibility point. By Invariant 4, at point v, Quancurrent summarises $f(\sigma')$, and, similarly, summarises the stream $A = \mathcal{S}(f(\sigma))$. Therefore, The query Q returns a value equal to the value returned by a sequential Quantiles sketch after processing the stream $A = \mathcal{S}(f(\sigma'))$. ■

We have shown that each query in $f(\sigma)$ estimates all updates that happened before its invocation. Specifically, a query invocation at the end of a finite execution σ returns a value equal to the value returned by a sequential sketch after processing $A = \mathcal{S}(f(\sigma))$. By this, we have proven that $f(\sigma) \in SeqSketch$. ■

Next, we prove the second part. We show that for every execution σ, $f(\sigma)$ is an r-relaxation of $l(\sigma)$ for $r = 4kS + (N - S)b$.

The order between operations satisfies:

Lemma 6.2.4. *Given a finite execution σ of Quancurrent, and given an operation O (query or update) in $l(\sigma)$, for every query Q in $l(\sigma)$ such that Q happened before O in $l(\sigma)$, then Q happened before O in $f(\sigma)$:*

$$Q \prec_{l(\sigma)} O \Rightarrow Q \prec_{f(\sigma)} O$$

Proof. If O is a query then the proof is immediate since the visibility point and the linearization point of a query are equal. Else, O is an update. By definition, the linearization point of an update happens before its visibility point. As the linearization point and visibility point of a query Q are equal, it follows that if $Q \prec_{l(\sigma)} O$ then $Q \prec_{f(\sigma)} O$. ■

Note that as the query linearisation point is equal to its visibility point, all queries in $f(\sigma)$ will also be in $l(\sigma)$.

We give an upper bound on the number of updates in NUMA-local buffers.

Lemma 6.2.5. *Given a finite execution σ of Quancurrent, the maximum number of unpropagated update operations in all Gather&Sort units is $4kS$, where S is the number of NUMA nodes:*

$$|up\_suffix(\sigma)| \leq 4kS.$$

Proof. If an update is included in $up\_suffix(\sigma)$, by definition, it is included in one of the G&SBuffers. The size of each G&SBuffer is less than or equal to $2k$. By definition, if both arrays in a Gather&Sort unit are full, no update thread (pinned to the same node as the Gather&Sort unit) can copy his local buffer's elements. It follows that $|up\_suffix(\sigma)| \leq 4kS$. ■

Now, we give an upper bound on the number of updates in threads' local buffers.

Lemma 6.2.6. *Given a finite execution σ of Quancurrent, the number of unpropagated updates in the local buffer of a thread T_i is bounded by b,*

$$|up\_suffix_i(\sigma)| \leq b$$

Proof. If an update is included in $up\_suffix_i(\sigma)$, by definition, it is included in T_i local buffer. As the size of threads' local buffer is b, it follows that $|up\_suffix_i(\sigma)| \leq b$. Note that when the local buffer of thread T_i is full, it copies $items_buf_i$ to one of the G&SBuffers and the corresponding updates will not be included in $up\_suffix_i(\sigma)$. ■

To prove that $f(\sigma)$ is an $(4kS + (N - S)b)$-relaxation of $l(\sigma)$, first, we will show that $f(\sigma)$ comprised of all but at most $r = 4kS + (N - S)b$ updates invocations in $l(\sigma)$ and their responses.

Lemma 6.2.7. *Given a finite execution σ of Quancurrent,*

$$|f(\sigma)| \geq |l(\sigma)| - (4kS + (N - S)b)$$

Proof. The linearization $l(\sigma)$ contains all updates that have been inserted into a thread's local buffer. $f(\sigma)$ contains all updates with visibility points, i.e., updates that have been propagated into the first level of the global sketch. An update, made by thread T_i, without a visibility point is in T_i local buffer or in one of the NUMA-local buffers of the G&Sort unit that T_i is pinned to. By definition, updates without visibility points are the unpropagated updates in G&SBuffers and the unpropagated updates in the local buffer of each update thread. Each G&Sort unit has an owner thread that tries to batch update into the global sketch. As such, for N update threads, S update threads' local buffer is empty as each of them is an owner

thread of a G&Sort unit. Therefore,

$$|f(\sigma)| = |l(\sigma)| - \left(\sum_{i=1}^{N-S} |up\_suffix_i(\sigma)|\right) - |up\_suffix(\sigma)| \tag{6.1}$$

From Lemma 6.2.6, $|up\_suffix_i(\sigma)| \leq b$ and from Lemma 6.2.5, $|up\_suffix(\sigma)| \leq S \cdot 4k$. Therefore,

$$|f(\sigma)| \geq |l(\sigma)| - (4kS + (N-S)b) \tag{6.2}$$ ■

To complete the poof that $f(\sigma)$ is an $(4kS + (N-S)b)$-relaxation of $l(\sigma)$, we will show that each invocation in $f(\sigma)$ is preceded by all but at most $(4kS+(N-S)b)$ of the invocations that precede the same invocation in $l(\sigma)$.

Lemma 6.2.8. *Given a finite execution σ of Quancurrent, $f(\sigma)$ is an $(4kS + (N-S)b)$-relaxation of $l(\sigma)$.*

Proof. Let O be an operation in $f(\sigma)$ such that O is also in $l(\sigma)$. Let Ops be a collection of operations preceded O in $l(\sigma)$ but not preceded O in $f(\sigma)$, i.e.,

$$Ops = \{O' | O' \prec_{l(\sigma)} O \wedge O' \not\prec_{f(\sigma)} O\}. \tag{6.3}$$

By Lemma 6.2.4, as the query linearization point is equal to its visibility point, a query is in $f(\sigma)$ if and only if it is in $l(\sigma)$. Thus $Q \notin Ops$ i.e., Ops includes only update operations. Let σ^{pre} be the prefix of σ and let σ^{post} be the suffix of σ such that

$$l(\sigma) = \{\sigma^{pre}, O, \sigma^{post}\}. \tag{6.4}$$

From Lemma 6.2.7, $|f(\sigma^{pre})| \geq |l(\sigma^{pre})| - (4kS + (N-S)b))$. As $|f(\sigma^{pre})|$ is the number of updates preceded O in $f(\sigma^{pre})$, and $|l(\sigma^{pre})|$ is the number of updates preceded O in $l(\sigma^{pre})$, it follows that

$$\begin{aligned} |Ops| &= |l(\sigma^{pre})| - |f(\sigma^{pre})| && (6.5)\\ &\leq |l(\sigma^{pre})| - (|l(\sigma^{pre})| - (4kS + (N-S)b)) && (6.6)\\ &\leq (4kS + (N-S)b). && (6.7) \end{aligned}$$

Therefore, by Definition 2.1.2, $f(\sigma)$ is an $(4k + b(N-1))$-relaxation of $l(\sigma)$. ■

Finally, we have proven that given a finite execution σ of Quancurrent, $l(\sigma)$ is strongly linearizable, $f(\sigma) \in SeqSketch$ and $f(\sigma)$ is an $(4kS + (N-S)b)$-relaxation of $l(\sigma)$. We have proven Lemma 6.2.1.

Chapter 7

Conclusion and Open Questions

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, i.e., retaining a small error bound with reasonable query freshness. 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 engage concurrently 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.

Another direction for future work is using a Quantiles sketch to implement a search index. Index structures are used when fast data access is needed. Such data structures are critical in practical settings, where large amounts of underlying data are paired with high search volumes and with a high level of concurrency on the hardware side via tens or even hundreds of parallel threads.

Indexes have been more memory, cache, and/or CPU efficient in the past decades. It is common for the index structure always to be stored in the main memory, with the data itself sitting on disk. The B-Tree is a commonly used index structure, which returns the location of a value within a key sorted set.

Kraska et al. [30] suggested that traditional index structures can be enhanced, or even replaced, with learned models, including deep-learning models, termed *learned indexes*. Continuous function, describing the data distribution, can be used to build more efficient data structures or algorithms. As noted by Kraska et al. [30], indexes are models. For example, a B-Tree can be considered as a model which takes a key as an input and returns the position of a record within a sorted array. Instead of a B-Tree, they suggested a recursive model index. That is, build a hierarchy of models, where at each stage the model takes the key as an input

and based on it picks another model, until the final stage predicts the position.

Quantile sketches can summarize large amounts of data in sub-linear space complexity. Therefore, We can use Quantiles sketches to reduce the memory overhead of an index. Also, Quantiles sketch can be used to approximate the data distribution and optimize different types of index structures.

Appendix A

Holes Analysis Proofs

To show the bound on the expected number of holes, we present some claims.

In Section 4.1 we show in Equation 4.2 that

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

Consider the following functions:

$$f_1(i) \triangleq \left(\frac{1}{2}\right)^{jb+2i+1} \binom{jb+2i}{i}.$$
$$f_2(i) \triangleq \frac{1}{4} \cdot \frac{(jb+2i+2)(jb+2i+1)}{(jb+i+1)(i+1)}.$$
$$f_3(j) \triangleq \frac{1}{4} \cdot \frac{(jb+2b)(jb+2b-1)}{(jb+b)b}.$$

Claim A.1. *$f_2(i)$ is monotonically decreasing for $0 \leq i \leq b-1$ and for $j, b > 2 \in \mathbb{N}$.*

Proof.

$$f_2(i+1) = \frac{1}{4} \cdot \frac{(jb+2i+4)(jb+2i+3)}{(jb+i+2)(i+2)}.$$

We show that $\frac{f_2(i+1)}{f_2(i)} \leq 1$ for $0 \leq i \leq b-2$.

$$\frac{f_2(i+1)}{f_2(i)} = \frac{(jb+2i+4)}{(jb+2i+2)} \cdot \frac{(jb+2i+3)}{(jb+2i+1)} \cdot \frac{(jb+i+1)}{(jb+i+2)} \cdot \frac{(i+1)}{(i+2)} \leq 1$$

Claim A.2. *$f_3(j)$ is monotonically increasing for $j, b > 2 \in \mathbb{N}$.*

Proof.

$$f_3(j+1) = \frac{1}{4} \cdot \frac{(jb+3b)(jb+3b-1)}{(jb+2b)b}.$$

We show that $\frac{f_3(j+1)}{f_3(j)} \geq 1$ for $j, b > 2 \in \mathbb{N}$

$$\begin{aligned}\frac{f_3(j+1)}{f_3(j)} &= \underbrace{\frac{(jb+3b)}{(jb+2b)}}_{\geq 1} \cdot \underbrace{\frac{(jb+3b-1)}{(jb+2b-1)}}_{\geq 1} \cdot \frac{(jb+b)}{(jb+2b)} \\ &\geq \frac{(jb+b)}{(jb+2b)} \geq \frac{(jb+2b)}{(jb+2b)} \geq 1\end{aligned}$$

Claim A.3. *$f_1(i)$ is monotonically increasing for $i \in \{0, 1, \ldots, b-1\}$ and for $j, b \in \mathbb{N}$.*

Proof. For $b = 1$, $i \in 0$ and the claim is immediate.
For $b = 2$, $i \in 0, 1$ thus,

$$\begin{aligned}f_1(0) &= \left(\frac{1}{2}\right)^{2j+1} \\ f_1(1) &= \left(\frac{1}{2}\right)^{2j+3} \binom{2j+2}{1} \\ &= \left(\frac{1}{2}\right)^{2j+1} \cdot \left(\frac{1}{2}\right)^{2} \cdot (2j+2) \\ &= f_1(0) \cdot \frac{2j+2}{4} \geq f_1(0) \cdot \frac{2+2}{4} = f_1(0).\end{aligned}$$

For $b > 2$, we show that $f_1(i+1) \geq f_1(i)$ for all $0 \leq i \leq b-1$ and for $j, b \in \mathbb{N}$

$$\begin{aligned}f_1(i) &= \left(\frac{1}{2}\right)^{jb+2i+1} \binom{jb+2i}{i} \\ f_1(i+1) &= \left(\frac{1}{2}\right)^{jb+2i+3} \binom{jb+2i+2}{i+1} \\ &= \left(\frac{1}{2}\right)^{jb+2i+3} \cdot \frac{(jb+2i+2)!}{(jb+i+1)! \cdot (i+1)!} \\ &= f_1(i) \cdot \frac{1}{4} \cdot \frac{(jb+2i+2)(jb+2i+1)}{(jb+i+1)(i+1)}\end{aligned}$$

Claim A.1 shows that $f_2(i) = \frac{1}{4} \cdot \frac{(jb+2i+2)(jb+2i+1)}{(jb+i+1)(i+1)}$ is monotonically decreasing for $0 \leq i \leq b-1$. Denote by $f_3(j)$ to be $f_2(b-1)$,

$$f_3(j) \triangleq \frac{1}{4} \cdot \frac{(jb+2b)(jb+2b-1)}{(jb+b)b} \tag{A.1}$$

Claim A.2 shows that $f_3(j)$ is monotonically increasing.

$$f_3(1) = \frac{1}{4} \cdot \frac{(3b)(3b-1)}{2b^2} = \frac{9b-3}{8b} \underbrace{=}_{b \geq 3} \frac{9b-b}{8b} \geq 1$$

Therefore, $f_3(j) \geq 1$ for all $j \geq 1$. With this, we have shown that

$$f_1(i+1) = f_1(i) \cdot \frac{1}{4} \cdot \frac{(jb+2i+2)(jb+2i+1)}{(jb+i+1)(i+1)} \geq f_1(i)$$

Claim A.4. $g_1(b) \triangleq \frac{3b-1}{b}$ *bounded by 3.*

Proof.

$$g_1(b) = \frac{3b-1}{b} < \frac{3b}{b} < 3$$

Claim A.5. $g_2(b) \triangleq \frac{3b+1}{2b+1}$ *bounded by 1.5.*

Proof.

$$g_2(b) = \frac{3b+1}{2b+1} = \frac{(2b+1)+b}{2b+1} = 1 + \frac{b}{2b+1} < 1 + \frac{b}{2b} < 1.5$$

Claim A.6. $g_3(b) \triangleq \frac{(b+1)^2}{b^2}$ *is monotonically decreasing and for* $b = 12$, $g_3(b) = \left(\frac{13}{12}\right)^2$

Proof. The proof is immediate. ■

Claim A.7. $\left(\frac{1}{2}\right)^{b-1} \cdot \binom{jb+3b-2}{b-1} \leq \binom{jb+2b-2}{b-1}$ *for all* $b \in \mathbb{N}$ *and* $j \geq 1, j \in \mathbb{N}$.

Proof.

$$\left(\frac{1}{2}\right)^{b-1} \cdot \binom{jb+3b-2}{b-1} =$$
$$\left(\frac{1}{2}\right)^{b-1} \cdot \frac{(jb+3b-2)!}{(b-1)!(jb+2b-1)!} =$$
$$\left(\frac{1}{2}\right)^{b-1} \cdot \frac{(jb+2b-2+b)!}{(b-1)!(jb+b-1+b)!} =$$
$$\left(\frac{1}{2}\right)^{b-1} \cdot \frac{(jb+2b-2)!}{(b-1)!(jb+b-1)!} \cdot \frac{(jb+2b-2+1)\cdots(jb+2b-2+b)}{(jb+b-1+1)\cdots(jb+b-1+b)} =$$
$$\left(\frac{1}{2}\right)^{b-1} \cdot \binom{jb+2b-2}{b-1} \cdot \prod_{t=1}^{b} \frac{jb+2b-2+t}{jb+b-1+t} =$$
$$\binom{jb+2b-2}{b-1} \cdot \prod_{t=1}^{b} \frac{jb+2b-2+t}{2(jb+b-1+t)} =$$
$$\binom{jb+2b-2}{b-1} \cdot \underbrace{\prod_{t=1}^{b} \frac{jb+2b-2+t}{(jb+2b-2+t)+jb+t}}_{\leq 1} \leq \binom{jb+2b-2}{b-1}$$

for all $b, j \in \mathbb{N}$ such that $j \geq 1$. ■

Bibliography

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, et al. "Scuba: Diving into data at facebook". In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1057–1067.

[2] Yehuda Afek, Guy Korland, and Eitan Yanovsky. "Quasi-Linearizability: Relaxed Consistency for Improved Concurrency". In: vol. 6490. Dec. 2010, pp. 395–410.

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

[4] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. "Are lock-free concurrent algorithms practically wait-free?" In: *Journal of the ACM (JACM)* 63.4 (2016), pp. 1–20.

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

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

[7] Mihai Budiu, Parikshit Gopalan, Lalith Suresh, Udi Wieder, Han Kruiger, and Marcos K Aguilera. "Hillview: a trillion-cell spreadsheet for big data". In: *Proceedings of the VLDB Endowment* 12.11 (2019), pp. 1442–1457.

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

[9] Graham Cormode. "Sketch techniques for approximate query processing". In: *Foundations and Trends in Databases. NOW publishers* (2011), p. 15.

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

[11] Graham Cormode, S. Muthukrishnan, and Ke Yi. "Algorithms for Distributed Functional Monitoring". In: *ACM Trans. Algorithms* 7.2 (Mar. 2011). URL: https://doi.org/10.1145/1921659.1921667.

[12] Graham Cormode and Shan Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications". In: *Journal of Algorithms* 55.1 (2005), pp. 58–75.

[13] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. "Gigascope: A Stream Database for Network Applications". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: Association for Computing Machinery, 2003, pp. 647–651. URL: https://doi.org/10.1145/872757.872838.

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

[15] Lee Rhodes Daniel Ting Jonathan Malkin. *Practical Sketching for Production Systems*. https://datasketches.apache.org/docs/img/Community/KDD_sketching_tutorial_pt2.pdf. 26th ACM SIGKDD conference on knowledge discovery and data mining. Aug. 2020.

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

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

[18] Flurry. *Flurry*. https://www.flurry.com/. Accessed September 22, 2022.

[19] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. "Moment-based quantile sketches for efficient high cardinality aggregation queries". In: *Proceedings of the VLDB Endowment* 11.11 (2018), pp. 1647–1660.

[20] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin J Strauss. "How to summarize the universe: Dynamic maintenance of quantiles". In: *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier. 2002, pp. 454–465.

[21] Wojciech Golab, Lisa Higham, and Philipp Woelfel. "Linearizable Implementations Do Not Suffice for Randomized Distributed Computation". In: STOC '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 373–382. URL: https://doi.org/10.1145/1993636.1993687.

[22] Michael Greenwald and Sanjeev Khanna. "Space-Efficient Online Computation of Quantile Summaries". In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. Santa Barbara, California, USA: Association for Computing Machinery, 2001, pp. 58–66. URL: https://doi.org/10.1145/375663.375670.

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

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

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

[26] Stefan Heule, Marc Nunkesser, and Alexander Hall. "HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm". In: *Proceedings of the 16th International Conference on Extending Database Technology*. EDBT '13. Genoa, Italy: Association for Computing Machinery, 2013, pp. 683–692. URL: https://doi.org/10.1145/2452376.2452456.

[27] Regant Hung and Hingfung F Ting. "An $\Omega(\frac{1}{\epsilon}log\frac{1}{\epsilon})$ Space Lower Bound for Finding ϵ-Approximate Quantiles in a Data Stream". In: *International Workshop on Frontiers in Algorithmics*. Springer. 2010, pp. 89–100.

[28] Piotr Indyk. "Algorithms for Dynamic Geometric Problems over Data Streams". In: *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. STOC '04. Chicago, IL, USA: Association for Computing Machinery, 2004, pp. 373–380. URL: https://doi.org/10.1145/1007352.1007413.

[29] Zohar Karnin, Kevin Lang, and Edo Liberty. "Optimal Quantile Approximation in Streams". In: *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 2016, pp. 71–78.

[30] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 489–504. URL: https://doi.org/10.1145/3183713.3196909.

[31] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. "Approximate Medians and Other Quantiles in One Pass and with Limited Memory". In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. SIGMOD '98. Seattle, Washington, USA: Association for Computing Machinery, 1998, pp. 426–435. URL: https://doi.org/10.1145/276304.276342.

[32] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G. Lindsay. "Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets". In: *Proceedings of the 1999 ACM SIGMOD International Conference on Man-*

agement of Data. SIGMOD '99. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1999, pp. 251–262. URL: https://doi.org/10.1145/304182.304204.

[33] Charles Masson, Jee E Rim, and Homin K Lee. "DDSketch: a fast and fully-mergeable quantile sketch with relative-error guarantees". In: *Proceedings of the VLDB Endowment* 12.12 (2019), pp. 2195–2205.

[34] J.I. Munro and M.S. Paterson. "Selection and sorting with limited storage". In: *Theoretical Computer Science* 12.3 (1980), pp. 315–323. URL: https://www.sciencedirect.com/science/article/pii/0304397580900614.

[35] Viswanath Poosala, Yannis E Ioannidis, et al. "Estimation of query-result distribution and its application in parallel-join load balancing". In: *VLDB*. Vol. 96. 1996, pp. 3–6.

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

[37] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. "Fast Concurrent Data Sketches". In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 117–129. URL: https://doi.org/10.1145/3332466.3374512.

[38] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. "Medians and beyond: New Aggregation Techniques for Sensor Networks". In: *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*. SenSys '04. Baltimore, MD, USA: Association for Computing Machinery, 2004, pp. 239–249. URL: https://doi.org/10.1145/1031495.1031524.

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

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

[41] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. "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. 13. NIH Public Access. 2015, p. 2182.

[42] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. "Quantiles over Data Streams: An Experimental Study". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Com-

puting Machinery, 2013, pp. 737–748. URL: https://doi.org/10.1145/2463676.2465312.

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

[44] Fuheng Zhao, Sujaya Maiyya, Ryan Wiener, Divyakant Agrawal, and Amr El Abbadi. "KLL± Approximate Quantile Sketches over Dynamic Datasets". In: *Proc. VLDB Endow.* 14.7 (Apr. 2021), pp. 1215–1227. URL: https://doi.org/10.14778/3450980.3450990.

ביחס למקרה הסדרתי. אנו מביאים ניסויים המראים כי הסקיצה מהירה יותר מפתרונות אחרים המממשים סקיצה מקבילית. כמו- כן הסקיצה בעלת תוצאות עדכניות יותר בסדר גודל מאשר הסקיצה המקבילית העדכנית ביותר שידועה לנו נכון לזמן כתיבת עבודת מחקר זו.

הארכיטקטורה של הסקיצה המקבילית מבוססת על חציצה מקומית של ערכים, אשר מפועפעים לאחר מכן באצווה לסקיצה משותפת. הסקיצה מביאה לביטול צוואר הבקבוק שנובע בעיקר מהצורך במיון חוצצים גדולים. המיון בסקיצה המקבילית מתבצע בשלוש רמות והסקיצה המשותפת עצמה מאורגנת במספר רמות אשר עשויות להיות מופצות וממוינות במקביל. שאילתות מטופלות כל הזמן במקביל לפעולות עדכון. כדי לאפשר סקילביליות של שאילתות, הסקיצה משרתת אותן מתמונת מצב של הסקיצה המשותפת השמורה במטמון מקומי תוך שמירה על רעננות התוצאות.

בעבודה זו אני מגדירים רשמית את מודל המערכת ומביאים הוכחה פורמאלית לנכונות הסקיצה המקבילית.

תקציר

אנאליזה בזמן אמת של זרם נתונים גדול הינה משימה הכרחית בעולם האנאליטיקות של ביג דאטה (נתוני עתק). מערכות מודרניות מטפלות כל הזמן בכמויות עצומות של מידע המגיע בקצב גבוה תוך כדי שהן מספקות ניתוחי זמן אמת עם זמני עיכוב מינימאליים. שרתי אינטרנט מתמודדים עם פטות של בתים ועליהם להיות תמיד זמינים, אינטראקטיביים ומהירים.

בעת ניתוח מידע בקנה מידע מאסיבי, חישוב מדויק, אפילו של שאילתות בסיסיות מאוד, עשוי לדרוש משאבי מחשוב עצומים (זיכרון וזמן חישוב). זה מוביל לפגיעה ביכולתה של מערכת להתמודד עם עומסים הולכים וגדלים. דוגמאות לשאילתות שניתן לשאול על המידע שנאסף כוללות שאילתת ספירת ערכים ייחודים, חישוב שברונים ואחוזונים, זיהוי תדר תכיפות ועוד. השיטות המדויקות לחישוב שאילתות אלה לא מצליחות להתאים את עצמן לכמויות המידע הממשיכות כל הזמן לגדול. כלים המסייעים לזהות מגמות או דפוסים מרכזיים של המידע מתוך נפח גדול של נתונים גולמיים הם בעלי ערך רב.

סקיצות הן מבני נתונים אשר מכילות מידע מקורב על זרם נתונים גדול. הן שייכות למשפחה של אלגוריתמי סטרימינג (אלגוריתמים הפועלים על זרם מידע גדול שתמיד נכנס) הנמצאים בשימוש נרחב בעולם הביג דאטה לביצוע ניתוח נתונים בקצב ובנפח גבוה. אלגוריתמים אלה מעבדים מערך נתונים מסיבי במעבר יחיד, הם מחשבים סיכומים קטנים מאוד (הנקראים סקיצות) של המידע ומאפשרים לבצע ניתוח מהיר בזמן אמת.

סוג סקיצה שימושי בעולם הביג דאטה לביצוע ניתוח מהיר בזמן היא סקיצת שברונים. שברון של זרם נתונים היא נקודת חתך מתחתיה נמצאים כל הערכים שקטנים מהערך בנקודה זו. לדוגמא, השברון ה-0.5 הוא החציון. הבנת אופן התפלגות המידע היא משימה בסיסית בניהול וניתוח נתונים, המשמשת לניטור מערכות ואפליקציות. סקיצת שברונים מעריכה את התפלגות הנתונים של זרם גדול כך ששאילתה לחישוב שברון מחזירה אומדן של הערך המדויק. כמו סקיצות אחרות, גם סקיצת שברונים היא בגודל תת-ליניארי והאומדנים שלה כנראה נכונים בקירוב. היא מספקת קירוב בתוך גבולות שגיאה מוכחים מתמטית עם הסתברות כשל מוגבלת. היא מאפשרת להפיק תוצאות עבור שאילתות (כגון, מהו האחוזון ה- 95) מהר יותר בכמה סדרי גודל.

בעבודה זו אנו מציגים סקיצת שברונים מקבילית וסקילבילית המתאימה למערכות המבצעות אנאליטיקות בזמן אמת תוך שמירה על גבולות שגיאה קטנים ותוצאות שאילתות מעודכנות.

הסקיצה בעלת תפוקה הגדלה ליניארית ככל שמספר התהליכים גדל. מתקבלת האצה של פי 12 עבור פעולות עדכון (הוספת ערך לסקיצה) והאצה של פי 30 עבור ביצוע שאילתות, עם 32 תהליכים,

המחקר בוצע בהנחייתו של פרופסור עדית קידר, בפקולטה להנדסת חשמל ומחשבים.

תודות

בראש ובראשונה, אני רוצה להודות לדיקן ולפרופסור עידית קידר, המנחה שלי לתזה לתואר שני. זה היה כבוד וזכות לעבוד תחת הדרכתך. תודה שהאמנת בי ונתת לי את ההזדמנות ללמוד ממך, זה היה מסע מדהים ובלתי יסולא בפז. ככזה, אני אסירת תודה לא פחות לעמיתי למחקר, הדוקטורנט אריק רינברג. תודה שהיית העוגן הקבוע, שאליו אני תמיד יכולה לפנות אליו לרעיונות, לשוחח איתו וללמוד ממנו. העזרה שלך עשתה את כל ההבדל. לחברי קבוצת המחקר של פרופסור עידית: גל אסא, עודד נאור, שיר כהן ורמי פאח'ורי – הזמן, הנוכחות, הרעיונות והתמיכה המתמדת שלכם היו הכרחיים, תודה לכם על כך. ברצוני להודות למשפחתי ולחבריי על האהבה והתמיכה המתמשכת. במיוחד, אני רוצה להודות לבעלי שגיא על התמיכה האינסופית, האהבה והעידוד להשלמת המסע האקדמי שלי. תודה שתמיד גילית עניין בעבודה שלי ועזרת לי להפיק את המיטב. תודה חייבת ללכת גם להוריי על כל האהבה והעזרה, על שתמיד האמינו בי והיו שם בשבילי.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

Quancurrent:
סקיצה מקבילית לשערוך שברונים של שטף מידע

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים בהנדסת חשמל

שקד אליאס זדה

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
אלול התשפ״ב חיפה ספטמבר 2022

Quancurrent:
סקיצה מקבילית לשערוך שברונים של שטף מידע

שקד אליאס זדה