Fast Concurrent Data Sketches

Arik Rinberg
arikrinberg@campus.technion.ac.il
Technion

Alexander Spiegelman
spiegelmans@vmware.com
VMware Research

Edward Bortnikov
ebortnik@verizonmedia.com
Yahoo Research

Eshcar Hillel
eshcar@verizonmedia.com
Yahoo Research

Idit Keidar
idish@ee.technion.ac.il
Technion and Yahoo Research

Lee Rhodes
lrhodes@verizonmedia.com
Verizon Media

Hadar Serviansky
hadar.serviansky@weizmann.ac.il
Weizmann Institute

Abstract

Data sketches are approximate succinct summaries of long data streams. They are widely used for processing massive amounts of data and answering statistical queries about it. Existing libraries producing sketches are very fast, but do not allow parallelism for creating sketches using multiple threads or querying them while they are being built. We present a generic approach to parallelising data sketches efficiently and allowing them to be queried in real time, while bounding the error that such parallelism introduces. Utilising relaxed semantics and the notion of strong linearisability we prove our algorithm's correctness and analyse the error it induces in two specific sketches. Our implementation achieves high scalability while keeping the error small. We have contributed one of our concurrent sketches to the open-source data sketches library.

1 Introduction

Data sketching algorithms, or *sketches* for short [15], have become an indispensable tool for high-speed computations over massive datasets in recent years. Their applications include a variety of analytics and machine learning use cases, e.g., data aggregation [9, 12], graph mining [14], anomaly (e.g., intrusion) detection [27], real-time data analytics [18], and online classification [25].

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 stream that approximates a specific query (unique element count, quantile values, etc.). The approximation is typically very accurate – the error drops fast with the stream size [15].

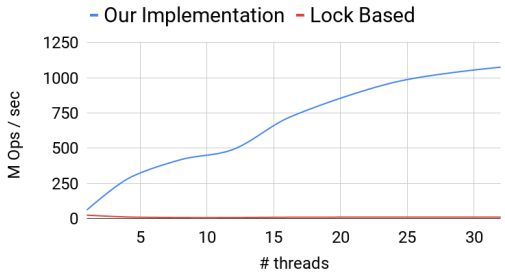


Figure 1. Scalability of DataSketches' Θ sketch protected by a lock vs. our concurrent implementation.

Practical sketch implementations have recently emerged in toolkits [3] and data analytics platforms (e.g., PowerDrill [21], Druid [18], Hillview [6], and Presto [2]). However, these implementations are not thread-safe, allowing neither parallel data ingestion nor concurrent queries and updates; concurrent use is prone to exceptions and gross estimation errors. Applications using these libraries are therefore required to explicitly protect all sketch API calls by locks [4, 7].

We present a generic approach to parallelising data sketches efficiently while bounding the error that such a parallelisation might introduce. Our goal is to enable simultaneous queries and updates to a sketch from multiple threads. Our solution is carefully designed to do so without slowing down operations as a result of synchronisation. This is particularly challenging because sketch libraries are extremely fast, often processing tens of millions of updates per second.

We capitalise on the well-known sketch *mergeability* property [15], which enables computing a sketch over a stream by merging sketches over substreams. Previous works have exploited this property for distributed stream processing (e.g., [16, 21]), devising solutions with a sequential bottleneck at the merge phase and where queries cannot be served before all updates complete. In contrast, our method is based on shared memory and constantly propagates results to a

This work was partially supported by the Technion Funds for Security Research.

PPoPP '20, February 22–26, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-6818-6/20/02…$15.00
https://doi.org/10.1145/3332466.3374512

queryable sketch. We adaptively parallelise stream processing: for small streams, we forgo parallel ingestion as it might introduce significant errors; but as the stream becomes large, we process it in parallel using small thread-local sketches with continuous background propagation of local results to the common (queryable) sketch.

We instantiate our generic algorithm with two popular sketches from the open-source Apache DataSketches (Incubating) library [3]: (1) a KMV Θ sketch [12], which estimates the number of unique elements in a stream; and (2) a Quantiles sketch [9] estimating the stream element with a given rank. We have contributed the former back to the Apache DataSketches (Incubating) library [5]. Yet we emphasize that our design is generic and applicable to additional sketches.

Figure 1 compares the ingestion throughput of our concurrent Θ sketch to that of a lock-protected sequential sketch, on multi-core hardware. As expected, the trivial solution does not scale whereas our algorithm scales linearly.

Concurrency induces an error, and one of the main challenges we address is analysing this additional error. To begin with, our concurrent sketch is a concurrent data structure, and we need to specify its semantics. We do so using a flavour of *relaxed consistency* similar to [10, 20] that allows operations to "overtake" some other operations. Thus, a query may return a result that reflects all but a bounded number of the updates that precede it. While relaxed semantics were previously used for data structures such as stacks [20] and priority queues [11, 23], we believe that they are a natural fit for data sketches. This is because sketches are typically used to summarise streams that arise from multiple real-world sources and are collected over a network with variable delays, and so even if the sketch ensures strict semantics, queries might miss some real-world events that occur before them. Additionally, sketches are inherently approximate. Relaxing their semantics therefore "makes sense", as long as it does not excessively increase the expected error. If a stream is much longer than the relaxation bound, then indeed the error induced by the relaxation is negligible. But since the error allowed by such a relaxation is additive, in small streams it may have a large impact. This motivates our adaptive solution, which forgoes relaxing small streams.

We proceed to show that our algorithm satisfies relaxed consistency. But this raises a new difficulty: relaxed consistency is defined with regards to a deterministic specification, whereas sketches are randomised. We therefore first de-randomise the sketch's behaviour by delegating the random coin flips to an oracle. We can then relax the resulting sequential specification. Next, because our concurrent sketch is used within randomised algorithms, it is not enough to prove its linearisability. Rather, we prove that our generic concurrent algorithm instantiated with sequential sketch S satisfies *strong linearisability* [19] with regards to a relaxed sequential specification of the de-randomised S.

We then analyse the error of the relaxed sketches under random coin flips, with an adversarial scheduler that may delay operations in a way that maximises the error. We show that our concurrent Θ sketch's error is coarsely bounded by twice that of the corresponding sequential sketch. The error of the concurrent Quantiles sketch approaches that of the sequential one as the stream size tends to infinity.

Main contribution In summary, this paper tackles the problem of concurrent sketches, offers a general efficient solution for it, and rigorously analyses this solution. While the paper makes use of many known techniques, it combines them in a novel way. The main technical challenges we address are (1) devising a high-performance generic algorithm that supports real-time queries concurrently with updates without inducing an excessive error; (2) proving the relaxed consistency of the algorithm; and (3) bounding the error induced by the relaxation in both short and long streams.

The paper proceeds as follows: Section 2 lays out the model for our work and Section 3 provides background on sequential sketches. In Section 4 we formulate a flavour of relaxed semantics appropriate for data sketches. Section 5 presents our generic algorithm, and Section 6 analyses error bounds. Section 7 empirically studies the Θ sketch's performance and error with different stream sizes. Finally, Section 8 concludes. The full paper [24] formally proves strong linearisability of our generic algorithm, and includes some of the mathematical derivations used in our analysis.

2 Model

We consider a non-sequentially consistent shared memory model that enforces program order on all variables and allows definition of *atomic* variables as in Java [1] and C++ [13]. Practically speaking, reads and writes of atomic variables are guarded by memory fences, which guarantee that all writes executed before a write W to an atomic variable are visible to all reads that follow (on any thread) a read R of the same atomic variable s.t. R occurs after W.

A thread takes *steps* according to a deterministic *algorithm* defined as a state machine. An *execution* of an algorithm is an alternating sequence of steps and states, where each step follows some thread's state machine. Algorithms implement objects supporting *operations*, such as query and update. An operation's execution consists of a series of steps, beginning with an *invoke* and ending in a *response*. The *history* of an execution σ, denoted $\mathcal{H}(\sigma)$, is its subsequence of operation invoke and response steps. In a *sequential history*, each invocation is immediately followed by its response. The *sequential specification (SeqSpec)* of an object is its set of allowed sequential histories.

A *linearisation* of a concurrent execution σ is a history $H \in$*SeqSpec* such that (1) after adding responses to some

pending invocations in σ and removing others, H and σ consist of the same invocations and responses (including parameters) and (2) H preserves the order between non-overlapping operations in σ. Golab et al. [19] have shown that in order to ensure correct behaviour of randomised algorithms under concurrency, one has to prove *strong linearisability*:

Definition 2.1 (Strong linearisability)**.** A function f mapping executions to histories is *prefix preserving* if for every two executions σ, σ' s.t. σ is a prefix of σ', $f(\sigma)$ is a prefix of $f(\sigma')$.

An algorithm A is a strongly linearisable implementation of an object o if there is a prefix preserving function f that maps every execution σ of A to a linearisation H of σ.

For example, executions of atomic variables are strongly linearisable.

3 Background: sequential sketches

A sketch S summarises a collection of elements $\{ a_1, a_2, \ldots, a_n \}$, processed in some order given as a stream $A = a_1, a_2, \ldots, a_n$. The desired summary is agnostic to the processing order, but the underlying data structures may differ due to the order. Its API is:

- S**.init()** initialises S to summarise the empty stream;
- S**.update(**a**)** processes stream element a;
- S**.query(**arg**)** returns the function estimated by the sketch over the stream processed thus far, e.g., the number of unique elements; takes an optional argument, e.g., the requested quantile.
- S**.merge(**S'**)** merges sketches S and S' into S; i.e., if S initially summarised stream A and S' summarised A', then after this call, S summarises the concatenation of the two, $A||A'$.

Example:* Θ *sketch Our running example is a Θ sketch based on the *K Minimum Values (KMV)* algorithm [12] given in Algorithm 1 (ignore the last three functions for now). It maintains a *sampleSet* and a parameter Θ that determines which elements are added to the sample set. It uses a random hash function h whose outputs are uniformly distributed in the range $[0, 1]$, and Θ is always in the same range. An incoming stream element is first hashed, and then the hash is compared to Θ. In case it is smaller, the value is added to *sampleSet*. Otherwise, it is ignored.

Because the hash outputs are uniformly distributed, the expected proportion of values smaller than Θ is Θ. Therefore, we can estimate the number of unique elements in the stream by dividing the number of (unique) stored samples by Θ (assuming that the random hash function is drawn independently of the stream values).

KMV Θ sketches keep constant-size sample sets: they take a parameter k and keep the k smallest hashes seen so far. Θ is 1 during the first k updates, and subsequently it is the hash of the largest sample in the set. Once the sample set is full, every update that inserts a new element also removes the largest one and updates Θ. This is implemented efficiently using a min-heap. The merge method adds a batch of samples to *sampleSet*.

Accuracy Today, sketches are used sequentially, so that the entire stream is processed and then S.query(arg) returns an estimate of the desired function on the entire stream. Accuracy is defined in one of two ways. One approach analyses the *Relative Standard Error (RSE)* of the estimate, formally defined in the full paper [24], which is the standard error normalized by the quantity being estimated. For example, a KMV Θ sketch with k samples has an RSE of less than $1/\sqrt{k-2}$ [12].

A *probably approximately correct (PAC)* sketch provides a result that estimates the correct result within some error bound ϵ with a failure probability bounded by some parameter δ. For example, a Quantiles sketch approximates the ϕth quantile of a stream with n elements by returning an element whose rank is in $[(\phi - \epsilon)n, (\phi + \epsilon)n]$ with probability at least $1 - \delta$ [9].

4 Relaxed consistency for concurrent sketches

Previous work by Alistarh et al. [10] has presented a formalisation for a randomized relaxation of an object. The main idea is to have the parallel execution approximately simulate the object's correct sequential behaviour, with some provided error distribution. In their framework, one considers the parallel algorithm and bounds the probability that it induces a large error relative to the deterministic sequential specification. This approach is not suitable for our analysis, since the sequential object we parallelise (namely the sketch) is itself randomised. Thus, there are two sources of error: (1) the approximation error in the sequential sketch and (2) the additional error induced by the parallelisation. For the former, we wish to leverage the existing literature on analysis of sequential sketches. To bound the latter, we use a different methodology: we first derandomise the sequential sketch by delegating its coin flips to an oracle, and then analyse the relaxation of the (now) deterministic sketch. Finally, we leverage the sequential sketch analysis to arrive at a distribution for the returned value of a query.

We adopt a variant of Henzinger et al.'s [20] *out-of-order* relaxation, which generalises quasi-linearisabilty [8]. Intuitively, this relaxation allows a query to "miss" a bounded number of updates that precede it. Because a sketch is order agnostic, we further allow re-ordering of the updates "seen" by a query.

A relaxed property for an object o is an extension of its sequential specification to allow more behaviours. This requires o to have a sequential specification, so we convert sketches into deterministic objects by capturing their randomness in an external oracle; given the oracle's output, the

sketches behave deterministically. For the Θ sketch, the oracle's output is passed as a hidden variable to *init*, where the sketch selects the hash function. In the Quantiles sketch, a coin flip is provided with every update. For a derandomised sketch, we refer to the set of histories arising in its sequential executions as *SeqSketch*, and use SeqSketch as its sequential specification. We can now define our relaxed semantics:

Definition 4.1 (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 SeqSketch is the set of histories that have r-relaxations in SeqSketch:

$SeqSketch^r \triangleq \{H' | \exists H \in \text{SeqSketch s.t. } H \text{ is an r-relaxation of } H'\}$.

Note that our formalism slightly differs from that of [20] in that we start with a serialisation H' of an object's execution that does not meet the sequential specification and then "fix" it by relaxing it to a history H in the sequential specification. In other words, we relax history H' by allowing up to r updates to "overtake" every query, so the resulting relaxation H is in SeqSketch.

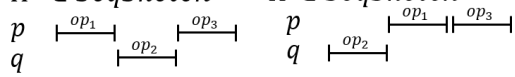


Figure 2. H is a 1-relaxation of H'.

An example is given in Figure 2, where H is a 1-relaxation of history H'. Both H and H' are sequential, as the operations don't overlap.

The impact of the r-relaxation on the sketch's error depends on the *adversary*, which may select up to r updates to hide from every query. There exist two adversary models: A *weak adversary* decides which r operations to omit from every query without observing the coin flips. A *strong adversary* may select which updates to hide after learning the coin flips. Neither adversary sees the protocol's internal state, however both know the algorithm and see the input. As the strong adversary knows the coin flips, it can then extrapolate the state; the weak adversary, on the other hand, cannot.

5 Generic concurrent sketch algorithm

We now present our generic concurrent algorithm. The algorithm uses, as a building block, an existing (non-parallel) sketch. To this end, we extend the standard sketch interface in Section 5.1, making it usable within our generic framework. Our algorithm is adaptive – it serialises ingestion in small streams and parallelises it in large ones. For clarity of presentation, we present in Section 5.2 the parallel phase of the algorithm, which provides relaxed semantics appropriate for large streams; in the full paper [24] we prove that it is strongly linearisable with respect to an r-relaxation of the sequential sketch with which it is instantiated. Section 5.3 then discusses the adaptation for small streams.

5.1 Composable sketches

In order to be able to build upon an existing sketch S, we first extend it to support a limited form of concurrency. Sketches that support this extension are called *composable*.

A composable sketch has to allow concurrency between merges and queries. To this end, we add a *snapshot* API that can run concurrently with merge and obtains a queryable copy of the sketch. The sequential specification of this operation is as follows:

S**.snapshot()** returns a copy S' of S such that immediately after S' is returned, S.query(arg) = S'.query(arg) for every possible arg.

A composable sketch needs to allow concurrency only between snapshots and other snapshot and merge operations, and we require that such concurrent executions be strongly linearisable. Our Θ sketch, shown below, simply accesses an atomic variable that holds the query result. In other sketches snapshots can be achieved efficiently by a double collect of the relevant state.

Pre-filtering When multiple sketches are used in a multi-threaded algorithm, we can optimise them by sharing "hints" about the processed data. This is useful when the stream sketching function depends on the processed stream prefix. For example, we explain below how Θ sketches sharing a common value of Θ can sample fewer updates. Another example is reservoir sampling [26]. To support this optimisation, we add the following two APIs:

S**.calcHint()** returns a value $h \neq 0$ to be used as a hint.

S**.shouldAdd(**h**,** a**)** given a hint h, filters out updates that do not affect the sketch's state.

Formally, the semantics of these APIs are defined using the notion of summary: (1) When a sketch is initialised, we say that its state (or simply the sketch) *summarises* the empty history, and similarly, the empty stream; we refer to the sketch as *empty*. (2) After we apply a sequential history

$$H = S.update(a_1), S.resp(a_1), \ldots S.update(a_n), S.resp(a_n)$$

to a sketch S, we say that S *summarises* history H, and, similarly, summarises the stream $a_1, \ldots, a_n$. Given a sketch S that summarises a stream A, if shouldAdd($S.calcHint()$, a) returns false then for every streams B_1, B_2 and sketch S' that summarises $A||B_1||a||B_2$, S' also summarises $A||B_1||B_2$.

These APIs do not need to support concurrency, and may be trivially implemented by always returning $true$. Note that S.shouldAdd is a static function that does not depend on the current state of S.

Composable Θ sketch We add the three additional APIs to Algorithm 1. The snapshot method copies *est*. Note that the result of a merge is only visible after writing to est, because it is the only variable accessed by the query. As *est* is an atomic variable, the requirement on snapshot and merge is met. To minimise the number of updates, calcHint returns Θ and shouldAdd checks if $h(a) < \Theta$, which is safe because the value of Θ in sketch S is monotonically decreasing. Therefore, if $h(a) \geq \Theta$ then $h(a)$ will never enter the *sampleSet*.

Algorithm 1 Composable Θ sketch.

```
1: variables
2:     sampleSet, init k 1's                          ▷ samples
3:     Θ, init 1                                      ▷ threshold
4:     atomic est, init 0                             ▷ estimate
5:     h, init random uniform hash function
6: procedure QUERY(arg)
7:     return est
8: procedure UPDATE(arg)
9:     if h(arg) ≥ Θ then return
10:    add h(arg) to sampleSet
11:    keep k smallest samples in sampleSet
12:    Θ ← max(sampleSet)
13:    est ← (|sampleSet| − 1) / Θ
14: procedure MERGE(S)
15:    sampleSet ← merge sampleSet and S.sampleSet
16:    keep k smallest values in sampleSet
17:    Θ ← max(sampleSet)
18:    est ← (|sampleSet| − 1) / Θ
19: procedure SNAPSHOT
20:    localCopy ← emptysketch
21:    localCopy.est ← est
22:    return localCopy
23: procedure CALCHINT
24:    return Θ
25: procedure SHOULDADD(H, arg)
26:    return h(arg) < H
```

5.2 Generic algorithm

To simplify the presentation and proof, we first discuss an unoptimised version of our generic concurrent algorithm (Algorithm 2 without the gray lines) called *ParSketch*, and later an optimised version of the same algorithm (Algorithm 2 including the gray lines and excluding underscored line 124).

The algorithm is instantiated by a composable sketch and sequential sketches. It uses multiple threads to process incoming stream elements and services queries at any time during the sketch's construction. Specifically, it uses N worker threads, $t_1, \ldots, t_N$, each of which samples stream elements into a local sketch $localS_i$, and a propagator thread t_0 that merges local sketches into a shared composable sketch *globalS*. Although the local sketch resides in shared memory, it is updated exclusively by its owner update thread t_i and read exclusively by t_0. Moreover, updates and reads do not happen in parallel, and so cache invalidations are minimised. The global sketch is updated only by t_0 and read by query threads. We allow an unbounded number of query threads.

After b updates are added to $localS_i$, t_i signals to the propagator to merge it with the shared sketch. It synchronises with t_0 using a single *atomic* variable $prop_i$, which t_i sets to 0. Because $prop_i$ is atomic, the memory model guarantees that all preceding updates to t_i's local sketch are visible to the background thread once $prop_i$'s update is. This signalling is relatively expensive (involving a memory fence), but we do it only once per b items retained in the local sketch.

After signalling to t_0, t_i waits until $prop_i \neq 0$ (line 125); this indicates that the propagation has completed, and t_i can reuse its local sketch. Thread t_0 piggybacks the hint H it obtains from the global sketch on $prop_i$, and so there is no need for further synchronisation in order to pass the hint.

Before updating the local sketch, t_i invokes shouldAdd to check whether it needs to process a or not. For example, the Θ sketch discards updates whose hashes are greater than the current value of Θ. The global thread passes the global sketch's value of Θ to the update threads, pruning updates that would end up being discarded during propagation. This significantly reduces the frequency of propagations and associated memory fences.

Query threads use the snapshot method, which can be safely run concurrently with merge, hence there is no need to synchronise between the query threads and t_0. The freshness of the query is governed by the r-relaxation. In the full paper [24], we prove Lemma 1 below, asserting that the relaxation is Nb. This may seem straightforward as Nb is the combined size of the local sketches. Nevertheless, proving this is not trivial because the local sketches pre-filter many additional updates, which, as noted above, is instrumental for performance.

Lemma 1. *ParSketch instantiated with SeqSketch is strongly linearisable with regards to* $SeqSketch^{Nb}$.

A limitation of *ParSketch* is that update threads are idle while waiting for the propagator to execute the merge. This may be inefficient, especially if a single propagator iterates through many local sketches. Algorithm 2 with the gray lines included and the underlined line omitted presents the optimised *OptParSketch* algorithm, which improves thread utilisation via double buffering.

In *OptParSketch*, $localS_i$ is an array of two sketches. When t_i is ready to propogate $localS_i[cur_i]$, it flips the cur_i bit denoting which sketch it is currently working on (line 126), and immediately sets $prop_i$ to 0 (line 129) in order to allow the propagator to take the information from the other one. It then starts digesting updates in a fresh sketch.

In the full paper [24] we prove the correctness of the optimised algorithm by simulating N threads of *OptParSketch* using $2N$ threads running *ParSketch*. We do this by showing a *simulation relation* [22]. We use forward simulation (with

Algorithm 2 Optimised generic concurrent algorithm.

```
101: variables
102:     composable sketch globalS, init empty
103:     constant b                                   ▷ relaxation is 2Nb
104:     for each update thread t_i , 0 ≤ i ≤ N
105:         sketch localS_i[2], init empty
106:         int cur_i, init 0
107:         int counter_i, init 0
108:         int hint_i, init 1
109:         int atomic prop_i, init 1
110: procedure PROPAGATOR
111:     while true do
112:         for all thread t_i s.t. prop_i = 0 do
113:             globalS.merge(localS_i[1-cur_i])
114:             localS_i[1-cur_i]←empty sketch
115:             prop_i ← globalS.calcHint()
116: procedure QUERY(arg)
117:     localCopy ← globalS.snapshot(localCopy)
118:     return localCopy.query(arg)
119: procedure UPDATE_i(a)
120:     if ¬shouldAdd(hint_i, a) then return
121:     counter_i ← counter_i + 1
122:     localS_i[cur_i].update(a)
123:     if counter_i = b then
124:         prop_i ← 0                               ▷ In non-optimised version
125:         wait until prop_i ≠ 0
126:         cur_i ← 1 − cur_i
127:         hint_i ← prop_i
128:         counter_i ← 0
129:         prop_i ← 0                               ▷ In optimised version
```

no prophecy variables), ensuring strong linearisability. We conclude the following theorem:

Theorem 1. OptParSketch *instantiated with SeqSketch is strongly linearisable with regards to* SeqSketch^{2Nb}.

5.3 Adapting to small streams

By Theorem 1, a query can miss up to r updates. For small streams, the error induced by this can be very large. For example, the sequential Θ sketch answers queries with perfect accuracy in streams with up to k unique elements, but if $k < r$, the relaxation can miss *all* updates. In other words, while the additive error is guaranteed to be bounded by r, the relative error can be infinite.

To rectify this, we implement *eager propagation* for small streams, whereby update threads propagate updates immediately to the shared sketch instead of buffering them. Note that during the eager phase, updates are processed sequentially. Support for eager propagation can be added to Algorithm 2 by initialising b to 1 and having the propagator thread raise it to the desired buffer size once the stream exceeds some pre-defined length. The error analysis of the next section can be used to determine the adaptation point.

6 Deriving error bounds

Section 6.1 discusses the error introduced to the expected estimation and RSE of the KMV Θ sketch. Section 6.2 analyses the PAC Quantiles sketch. The full paper [24] contains mathematical derivations used throughout this section.

6.1 Θ error bounds

We bound the error introduced by an r-relaxation of the Θ sketch. Given Theorem 1, the optimised concurrent sketch's error is bounded by the relaxation's error bound for $r = 2Nb$. We consider strong and weak adversaries, $\mathcal{A}_s$ and $\mathcal{A}_w$, resp. For the strong adversary we are able to show only numerical results, whereas for the weak one we show closed-form bounds. The results are summarised in Table 1. Our analysis relies on known results from order statistics [17]. It focuses on long streams, and assumes $n > k + r$.

We would like to analyse the distribution of the k^{th} largest element in the stream that the relaxed sketch processes, as this determines the result returned by the algorithm. We cannot use order statistics to analyse this because the adversary alters the stream and so the stream seen by the algorithm is not random. However, the stream of hashed unique elements seen by the adversary *is* random. Furthermore, if the adversary hides from the algorithm j elements smaller than Θ, then the k^{th} largest element in the stream seen by the sketch is the $(k + j)^{th}$ largest element in the original stream seen by the adversary. This element is a random variable and therefore we can apply order statistics to it.

We thus model the hashed unique elements in the stream A processed before a given query as a set of n labelled iid random variables $A_1, \ldots, A_n$, taken uniformly from the interval $[0, 1]$. Note that A is the stream observed by the reference sequential sketch, and also by adversary that hides up to r elements from the relaxed sketch. Let $M_{(i)}$ be the i^{th} minimum value among the n random variables $A_1, \ldots, A_n$.

Let $est(x) \triangleq \frac{k-1}{x}$ be the estimate computation with a given $x = \Theta$ (line 18 of Algorithm 1). The sequential (non-relaxed) sketch returns $e = est(M_{(k)})$. It has been shown that the sketch is unbiased [12], i.e., $E[e] = n$ the number of unique elements, and $RSE[e] \leq \frac{1}{\sqrt{k-2}}$. The *Relative Standard Mean Error (RSME)* is the error relative to the mean, formally defined in the full paper [24]. Because this sketch is unbiased, $RSE[e] = RSME[e]$.

In a relaxed history, the adversary chooses up to r variables to hide from the given query so as to maximise its error. It can also re-order elements, but the state of a Θ sketch after a set of updates is independent of their processing order. Let $M^r_{(i)}$ be the i^{th} minimum value among the hashes seen by the query, i.e., arising in updates that precede the query in the relaxed history. The value of Θ is $M^r_{(k)}$, which is equal to $M_{(k+j)}$ for some $0 \leq j \leq r$. We do not know if the adversary can actually control j, but we know that it can impact it, and so for our error analysis, we consider strictly stronger adversaries – we

	Sequential sketch		Strong adversary $\mathcal{A}_s$	Weak adversary $\mathcal{A}_w$
	Closed-form	Numerical	Numerical	Closed-form
Expectation	n	2^{15}	$2^{15} \cdot 0.995$	$n\frac{k-1}{k+r-1}$
RSE	$\leq \frac{1}{\sqrt{k-2}}$	$\leq 3.1\%$	$\leq 3.8\%$	$\leq 2\frac{1}{\sqrt{k-2}}$

Table 1. Analysis of Θ sketch with numerical values for $r = 8, k = 2^{10}, n = 2^{15}$.

allow both the weak and the strong adversaries to choose the number of hidden elements j. Our error analysis gives an upper bound on the error induced by our adversaries. Note that the strong adversary can choose j based on the coin flips, while the weak adversary cannot, and therefore chooses the same j in all runs. In the full paper [24] we show that the largest error is always obtained either for $j = 0$ or for $j = r$.

Given an adversary $\mathcal{A}$ that induces an approximation $e_{\mathcal{A}}$, in the full paper [24] we prove the following bound:

$$\mathrm{RSE}[e_{\mathcal{A}}] \leq \sqrt{\frac{\sigma^2(e_{\mathcal{A}})}{n^2}} + \sqrt{\frac{(E[e_{\mathcal{A}}] - n)^2}{n^2}}.$$

Strong adversary $\mathcal{A}_s$ The strong adversary knows the coin flips in advance, and thus chooses j to be $g(0, r)$, where g is the choice that maximises the error:

$$g(j_1, j_2) \triangleq \arg\max_{j \in \{j_1, j_2\}} \left| \frac{k-1}{M_{(k+j)}} - n \right|.$$

In Figure 3 we plot the regions where g equals 0 and g equals r, based on their possible combinations of values. The estimate induced by $\mathcal{A}_s$ is $e_{\mathcal{A}_s} \triangleq \frac{k-1}{M_{(k+g(0,r))}}$. The expectation and standard error of $e_{\mathcal{A}_s}$ are calculated by integrating over the gray areas in Figure 3 using their joint probability function from order statistics. In the full paper [24] we give the formulas for the expected estimate and its RSE bound, resp. We do not have closed-form bounds for these equations. Example numerical results are shown in Table 1.

Weak adversary $\mathcal{A}_w$ Not knowing the coin flips, $\mathcal{A}_w$ chooses j that maximises the expected error for a random hash function: $E[n - est(M^r_{(k)})] = E[n - est(M_{(k+j)})] = n - n\frac{k-1}{k+j-1}$.

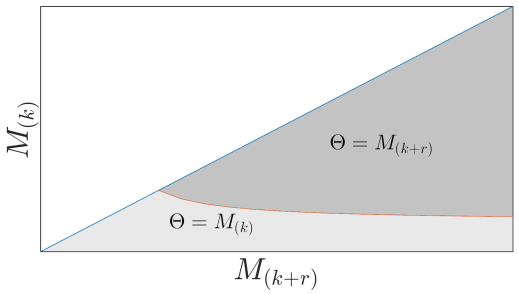


Figure 3. Areas of $M_{(k)}$ and $M_{(k+r)}$. In the dark gray $\mathcal{A}_s$ induces $\Theta = M_{(k+r)}$, and in the light gray, $\Theta = M_{(k)}$. The white area is not feasible.

Obviously this is maximised for $j = r$. The orange curve in Figure 4 depicts the distribution of $e_{\mathcal{A}_w}$, and the distribution of e is shown in blue.

In the full paper [24], we show that the RSE is bounded by $\sqrt{\frac{1}{k-2}} + \frac{r}{k-2}$ for $\hat{\mathcal{A}}_w$, and therefore so is that of $\mathcal{A}_w$. Thus, whenever r is at most $\sqrt{k-2}$, the RSE of the relaxed Θ sketch is coarsely bounded by twice that of the sequential one. And in case $k \gg r$, the addition to the *RSE* is negligible.

6.2 Quantiles error bounds

We now analyse the error for any implementation of the sequential Quantiles sketch, provided that the sketch is *PAC*, meaning that a query for quantile ϕ returns an element whose rank is between $(\phi - \epsilon)n$ and $(\phi + \epsilon)n$ with probability at least $1 - \delta$ for some parameters ϵ and δ. We show that the r-relaxation of such a sketch returns an element whose rank is in the range $(\phi \pm \epsilon_r)n$ with probability at least $1 - \delta$ for $\epsilon_r = \epsilon - \frac{r\epsilon}{n} + \frac{r}{n}$.

Although the desired summary is order agnostic here too, Quantiles sketch implementations (e.g., [9]) are sensitive to the processing order. In this case, advanced knowledge of the coin flips can increase the error already in the sequential sketch. Therefore, we do not consider a strong adversary, but rather discuss only the weak one. Note that the weak adversary attempts to maximise ϵ_r.

Consider an adversary that knows ϕ and chooses to hide i elements below the ϕ quantile and j elements above it, such that $0 \leq i + j \leq r$. The rank of the element returned by the query among the $n - (i + j)$ remaining elements is in the range $\phi(n - (i + j)) \pm \epsilon(n - (i + j))$. There are i elements below this quantile that are missed, and therefore its rank in

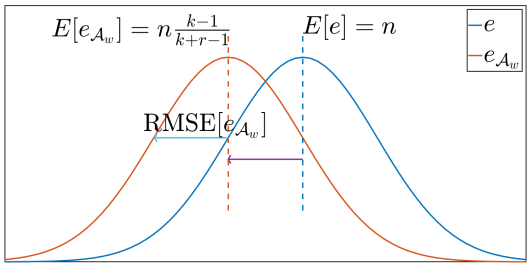


Figure 4. Distribution of estimators e and $e_{\mathcal{A}_w}$. The RSE of $e_{\mathcal{A}_w}$ with regards to n is bounded by the relative bias plus the RMSE of $e_{\mathcal{A}_w}$.

the original stream is in the range:

$$[(\phi-\epsilon)(n-(i+j))+i, (\phi+\epsilon)(n-(i+j))+i]\,. \quad (1)$$

This can be rewritten as:

$$\begin{aligned}&[\phi n-(\phi j-(1-\phi)i+\epsilon(n-(i+j))),\\ &\phi n+((1-\phi)i-\phi j+\epsilon(n-(i+j)))]\end{aligned} \quad (2)$$

In the full paper [24], we show that the r-relaxed sketch returns an element whose rank is between $(\phi-\epsilon_r)n$ and $(\phi+\epsilon_r)n$ with probability at least $1-\delta$, where $\epsilon_r = \epsilon - \frac{r\epsilon}{n} + \frac{r}{n}$. Thus the impact of the relaxation diminishes as n grows.

7 Θ sketch evaluation

This section presents an evaluation of an implementation of our algorithm for the Θ sketch. Section 7.1 presents the methodology for the analysis. Section 7.2 shows the results under different workloads and scenarios. Finally, Section 7.3 discusses the tradeoff between accuracy and throughput.

7.1 Setup and methodology

Our implementation [5] extends the code in Apache DataSketches (Incubating) [3], a Java open-source library of stochastic streaming algorithms. The Θ sketch there differs slightly from the KMV Θ sketch we used as a running example, and is based on a HeapQuickSelectSketch family. In this version, the sketch stores between k and $2k$ items, whilst keeping Θ as the k^{th} largest value. When the sketch is full, it is sorted and the largest k values are discarded.

Concurrent Θ sketch is generally available in the Apache DataSketches (Incubating) library since V0.13.0. The sequential implementation and the sketch at the core of the global sketch in the concurrent implementation are the both HeapQuickSelectSketch, which is the default sketch family.

As explained in Section 5.3, we implement a limit for eager propagation as a function of the configurable error parameter e; the function we use is $2/e^2$. The local sketches define b as a function of k, e, and N (the number of writer threads). The error induced by the relaxation does not exceed e, and thus the total error is bounded by $\max\{e+\frac{1}{\sqrt{k}}, \frac{2}{\sqrt{k}}\}$.

Eager propagation, as described in the pseudo-code, requires context switches incurring a high overhead. In the implementation, either the local thread itself executes every update to the global sketch (equivalent to a buffer size of 1) or lazily delegates updates to a background thread. While the sketch is in eager propagation mode, the global sketch is protected by a shared boolean flag. When the sketch switches to estimate mode it is guaranteed that no eager propagation gets through; instead local threads pass the buffer via lazy propagation. This implementation ensures that: (a) local threads avoid costly context switches when the sketch is small, and (b) lazy propagation by a background thread is done without synchronisation.

Unless otherwise stated, sketches are configured with $k = 4096$, and $e = 0.04$; thus the exact limit is $2/e^2 = 1250$, and b is set (by the implementation) to a value between 1 and 5 to accommodate the error bound. Our first set of tests run on a 12-core Intel Xeon E5-2620 machine – this machine is similar to that which is used by production servers. For the scalability evaluation (shown in the introduction) we use a 32-core Intel Xeon E5-4650 to get a large number of threads. Both machines have hyper-threading disabled, as it introduces non-monotonic effects among threads sharing a core.

We focus on two workloads: (1) write-only – updating a sketch with a stream of unique values; (2) mixed read-write workload – updating a sketch with background reads querying the number of unique values in the stream. Background reads refer to dedicated threads that occasionally (with 1ms pauses) execute a query. These workloads were chosen to simulate read-world scenarios where updates are constantly streaming from a feed or multiple feeds, while queries arrive at a lower rate.

To run the experiments we employ a multi-thread extension of the characterization framework. This is the Apache DataSketch evaluation benchmark suite, which measures both the speed and accuracy of the sketch.

For measuring write throughput, the sketch is fed with a continuous data stream. The size of the stream varies from 1 to 8M unique values. For each size x we measure the time t it takes to feed the sketch x unique values, and present it in term of throughput (x/t). To minimise measurement noise, each point on the graph represents an average of many trials. The number of trials is very high (2^{18}) for points at the low end of the graph. It gradually decreases as the size of the sketch increases. At the high end (at 8M values per trial) the number of trials is 16. This is because smaller stream sizes tend to suffer more from measurement noise.

The accuracy of a concurrent Θ sketch is measured only in a single-thread environment. As in the performance evaluations, the x-axis represents the number of unique values fed into the sketch by a single writing thread. For each size x, one trial logs the estimation result after feeding x unique values to the sketch. In addition, it logs the Relative Error (RE) of the estimate, where $RE = MeasuredValue/TrueValue - 1$. This trial is repeated 4K times, logging all estimation and RE results. The curves depict the mean and some quantiles of the distributions of error measured at each x-axis point on the graph, including the median. This type of graph is called a "pitchfork".

7.2 Results

Accuracy results Our first set of tests runs on a 12-core Intel Xeon E5-2620 machine. The accuracy results for the concurrent Θ sketch without eager propagation are presented in Figure 5a. There are two interesting phenomena worth noting. First, it is interesting to see empirical evaluation reflecting the theoretical analysis presented in Section 6.1, where the pitchfork is distorted towards underestimating

the number of unique values. Specifically, the mean relative error is smaller than 0 (showing a tendency towards underestimating), and the relative error in all measured quantiles tends to be smaller than the relative error of the sequential implementation.

Second, when the stream size is less than $2k$, $\Theta = 1$ and the estimation is the number of values propagated to the global sketch. If we forgo eager propagation, the number of values in the global sketch depends on the delay in propagation. The smaller the sketch, the more significant the impact of the delay, and the mean error reaches as high as 94% (the error in the figure is capped at 10%). As the number of propagated values approaches $2k$, the delay in propagation is less significant, and the mean error decreases. This excessive error is remedied by the eager propagation mechanism. The maximum error allowed by the system is passed as a parameter to the concurrent sketch, and the global sketch uses eager propagation to stay within the allowed error limit. Figure 5b depicts the accuracy results when applying eager propagation. The figures are similar when the sketch begins lazy propagation, and the error stays within the 0.04 limit as long as eager propagation is used.

Write-only workload Figure 6a presents throughput measurements for a write-only workload. The results are shown in loglog scale. Figure 6b zooms-in on the throughput of large streams.

When considering large stream sizes, the concurrent implementation scales with the number of threads, peaking at almost 300M operations per second with 12 threads. The performance of the lock-based implementation, on the other hand, degrades as the contention on the lock increases. Its peak performance is 25M operations per second with a single thread. Namely, with a single thread, the concurrent Θ sketch outperforms the lock-based implementation by 12x, and with 12 threads by more than 45x.

For small streams, wrapping a single thread with a lock is the most efficient method. Once the stream contains more than 200K unique values, using a concurrent sketch with 4 or more local threads is more efficient. The crossing point where a single local buffer is faster than the lock-based implementation is around 700K unique values.

Mixed workload Figure 7 presents the throughput measurements of a mixed read-write workload. We compare runs with a single updating thread and 2 updating threads (and 10 background reader threads). Although we see similar trends as in the write-only workload, the effect of background readers is more pronounced in the lock-based implementation than in the concurrent one; this is expected as the reader threads compete for the same lock as the writers. The peak throughput of a single writer thread in the concurrent implementation is 55M ops/sec both with and without background readers. The peak throughput of a single writer thread in the lock-based implementation degrades from 25M ops/sec without background reads to 23M ops/sec with them; this is an almost 10% slowdown in performance. Recall that in this scenario reads are infrequent, and so the degradation is not dramatic.

Scalability results To provide a better scalability analysis, we aim to maximize the number of threads working on the sketch. Therefore, we run this test on a larger machine – we use a 32-core Xeon E5-4650 processors. We ran an *update-only* workload in which a sketch is built from a very large stream, repeating each test 16 times.

In Figure 1 (in the introduction) we compare the scalability of our concurrent Θ sketch and the original sketch wrapped with a read/write lock in an update-only workload, for $b = 1$ and $k = 4096$. As expected, the lock-based sequential sketch does not scale, and in fact it performs worse when accessed concurrently by many threads. In contrast, our sketch achieves almost perfect scalability. Θ quickly becomes small enough to allow filtering out most of the updates and so the local buffers fill up slowly.

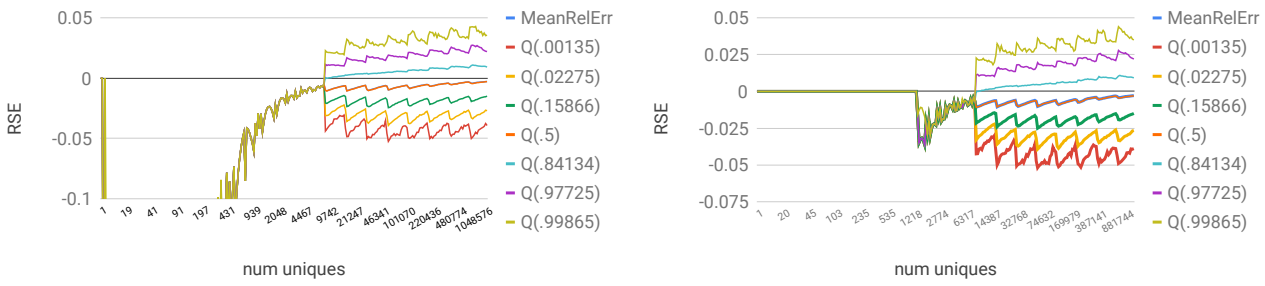
7.3 Accuracy-throughput tradeoff

The speedup achieved by eager propagation in small streams is presented in Figure 8. This is an additional advantage of eager propagation in small streams, beyond the accuracy benefit reported in Figure 5. The improvement is as high as 84x for tiny sketches, and tapers off as the sketch grows. The slowdown in performance when the sketch size exceeds $2k$ can be explained by the reduction in the local buffer size (from $b = 16$ to $b = 5$), needed in order to accommodate for the required error bound.

Next we discuss the impact of k. One way to increase the throughput of the concurrent Θ sketch is by increasing the size of the global sketch, namely increasing k. On the other hand, this change also increases the error of the estimate. Table 2 presents the tradeoffs between performance and accuracy. Specifically, it presents the crossing-point, namely the smallest stream size for which the concurrent implementation outperforms the lock-based implementation (both running a single thread). It further presents the maximum values (across all stream sizes) of the median error and 99th percentile error for a variety of k values. The table shows that as the sketch promises a smaller error (by using a larger k), a larger stream size is needed to justify using the concurrent sketch with all its overhead.

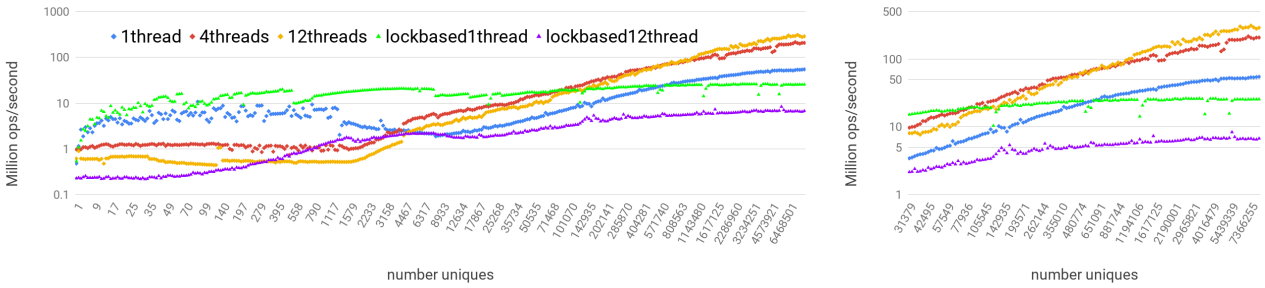
	thpt crossing point	mean error	error $Q = 0.99$
$k = 256$	15,000	0.16	0.27
$k = 1024$	100,000	0.05	0.13
$k = 4096$	700,000	0.03	0.05

Table 2. Performance vs accuracy as a function of k.



(a) No eager propagation ($e = 1.0$) **(b)** With eager propagation, error bound defined by $e = 0.04$

Figure 5. Concurrent Θ measured quantiles vs RSE, $k = 4096$.



(a) Throughput, loglog scale **(b)** Zooming-in on large sketches

Figure 6. Write-only workload, $k = 4096$, $e = 0.04$.

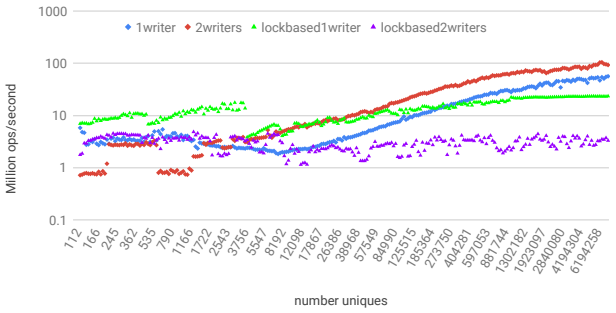


Figure 7. Mixed workloads: writers with background reads, $k = 4096$, $e = 0.04$.

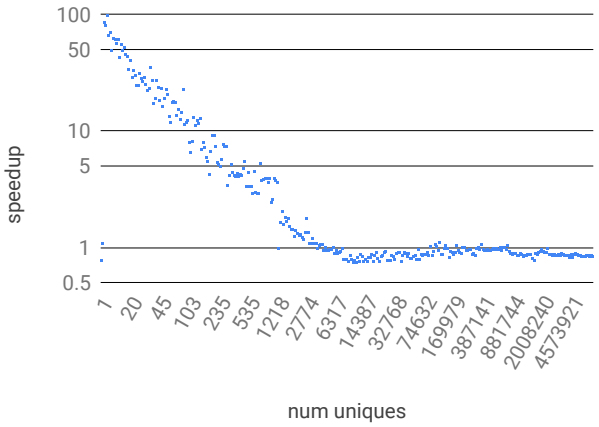


Figure 8. Throughput speedup of eager ($e = 0.04$) vs no-eager ($e = 1.0$) propagation, $k = 4096$.

8 Conclusions

Sketches are widely used by a range of applications to process massive data streams and answer queries about them. Library functions producing sketches are optimised to be extremely fast, often digesting tens of millions of stream elements per second. We presented a generic algorithm for parallelising such sketches and serving queries in real-time; the algorithm is strongly linearisable with regards to relaxed semantics. We showed that the error bounds of two representative sketches, Θ and Quantiles, do not increase drastically with such a relaxation. We also implemented and evaluated the solution, showed it to be scalable and accurate, and integrated it into the open-source Apache DataSketches (Incubating) library. While we analysed only two sketches, future work may leverage our framework for other sketches. Furthermore, it would be interesting to investigate additional uses of the hint, for example, in order to dynamically adapt the size of the local buffers and respective relaxation error.

References

[1] 2011. Java Language Specification: Chapter 17 - Threads and Locks. https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html.

[2] 2018. HyperLogLog in Presto: A significantly faster way to handle cardinality estimation. https://code.fb.com/data-infrastructure/hyperloglog/.
[3] 2019. Apache DataSketches (Incubating). https://incubator.apache.org/clutch/datasketches.html.
[4] 2019. ArrayIndexOutOfBoundsException during serialization. https://github.com/DataSketches/sketches-core/issues/178#issuecomment-365673204.
[5] 2019. DataSketches: Concurrent Theta Sketch Implementation. https://github.com/apache/incubator-datasketches-java/blob/master/src/main/java/org/apache/datasketches/theta/ConcurrentDirectQuickSelectSketch.java.
[6] 2019. Hillview: A Big Data Spreadsheet. https://research.vmware.com/projects/hillview.
[7] 2019. SketchesArgumentException: Key not found and no empty slot in table. https://groups.google.com/d/msg/sketches-user/S1PEAneLmhk/dI8RbN6iBAAJ..
[8] Yehuda Afek, Guy Korland, and Eitan Yanovsky. 2010. Quasi-linearizability: Relaxed Consistency for Improved Concurrency. In *Proceedings of the 14th International Conference on Principles of Distributed Systems (OPODIS'10)*. Springer-Verlag, Berlin, Heidelberg, 395–410. http://dl.acm.org/citation.cfm?id=1940234.1940273
[9] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. 2012. Mergeable Summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '12)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/2213556.2213562
[10] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z Li, and Giorgi Nadiradze. 2018. Distributionally linearizable data structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM.
[11] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. 2015. The SprayList: A Scalable Relaxed Priority Queue. *SIGPLAN Not.* 50, 8 (Jan. 2015), 11–20. https://doi.org/10.1145/2858788.2688523
[12] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. 2002. Counting Distinct Elements in a Data Stream. In *Randomization and Approximation Techniques in Computer Science*, Jos'e D. P. Rolim and Salil Vadhan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–10.
[13] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. *SIGPLAN Not.* 43, 6 (June 2008), 68–78. https://doi.org/10.1145/1379022.1375591
[14] Edith Cohen. 2014. All-distances Sketches, Revisited: HIP Estimators for Massive Graphs Analysis. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '14)*. ACM, New York, NY, USA, 88–99. https://doi.org/10.1145/2594538.2594546
[15] Graham Cormode. 2017. Data Sketching. *Queue* 15, 2, Article 60 (April 2017), 19 pages. https://doi.org/10.1145/3084693.3104030
[16] Graham Cormode, S Muthukrishnan, and Ke Yi. 2011. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms (TALG)* 7, 2 (2011), 21.
[17] Herbert Aron David and Haikady Navada Nagaraja. 2004. Order statistics. *Encyclopedia of Statistical Sciences* 9 (2004).
[18] Druid. [n.d.]. How We Scaled HyperLogLog: Three Real-World Optimizations. http://druid.io/blog/2014/02/18/hyperloglog-optimizations-for-real-world-systems.html.
[19] Wojciech Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*. ACM, 373–382.
[20] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. 2013. Quantitative relaxation of concurrent data structures. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 317–328.
[21] Stefan Heule, Marc Nunkesser, and Alex Hall. 2013. HyperLogLog in Practice: Algorithmic Engineering of a State of The Art Cardinality Estimation Algorithm. In *Proceedings of the EDBT 2013 Conference*. Genoa, Italy.
[22] Nancy A Lynch. 1996. *Distributed algorithms*. Elsevier.
[23] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2014. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. *arXiv preprint arXiv:1411.1209* (2014).
[24] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, and Hadar Serviansky. 2019. Fast Concurrent Data Sketches. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. ACM, 207–208.
[25] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. 2018. Sketching Linear Classifiers over Data Streams. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 757–772. https://doi.org/10.1145/3183713.3196930
[26] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.
[27] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 561–575. https://doi.org/10.1145/3230543.3230544

A Artifact Appendix

A.1 Abstract

The artifact contains all the JARs of version 0.12 of the DataSketches library, before it moved into Apache (Incubating), as well as configurations and shell scripts to run our tests. It can support the results found in the evaluated section of our PPoPP'2020 paper Fast Concurrent Data Sketches. To validate the results, run the test scripts and check the results piped in the according text output files.

A.2 Artifact check-list (meta-information)

- **Algorithm: Θ Sketch**
- **Program: Java code**
- **Compilation: JDK 8, and each package is compiled using maven**
- **Binary: Java executables**
- **Run-time environment: Java**
- **Hardware: Ubuntu on 12 core server and 32 core server with hyperthreading disabled**
- **Metrics: Throughput and accuracy**
- **Output: Runtime throughputs, and runtime accuracy**
- **How much time is needed to prepare workflow (approximately)?: Using precomipled packages, none.**
- **How much time is needed to complete experiments (approximately)?: Many hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: Apache License 2.0**

A.3 Description

A.3.1 How delivered

We have provided all the JAR files we used for running our tests, along with scripts. Meanwhile, the project has migrated to the

Apache DataSketches (Incubating) library, which is an open source project under Apache License 2.0, and is hosted with code, API specifications, build instructions, and design documentations on Github.

A.3.2 Hardware dependencies

Our tests require a 12-core Intel Xeon E5-2620 machine, and four Intel Xeon E5-4650 processors, each with 8 cores. Hyper-threading is disabled on both machines..

A.3.3 Software dependencies

Building and running the JAR files requires JDK 8; the files don't compile otherwise. To use the automated scripts, we require python3 and git to be installed. The Apache DataSketches (Incubating) library has been tested on Ubuntu 12.04/14.04, and is expected to run correctly under other Linux distributions.

A.4 Installation

First, clone the repository:

```
$ git clone https://github.com/ArikRinberg/FastConcurrentDataSketchesArtifact
```

We have provided the necessary JAR files for recreating our experiment in the repository.

A.5 Experiment workflow

1. After cloning the repository:

```
$ cd FastConcurrentDataSketchesArtifact
```

   In the current working directory, there should be the following JAR files:
   - memory-0.12.1.jar
   - sketches-core-0.12.1-SNAPSHOT.jar
   - characterization-0.1.0-SNAPSHOT.jar
2. Next, run the tests:

```
$ python3 run_test.py TEST
```

   Where TEST is one of the following: figure_1, figure_6_a, figure_6_b, figure_7, figure_8, figure_9, or table_2.
3. The results of each test will be in txt files in the current working directory, either SpeedProfile or AccuracyProfile:

   SpeedProfile: The txt file contains three columns: **InU** – the number of unique items (the x axis of most graphs), **Trials** – the number of trials for this run, **nS/u** – nano seconds per update. The y axis of the throughput graphs is given as updates per second, therefore a conversion is needed.

   AccuracyProfile: The txt file contains the columns corresponding to the figure legend, where **InU** is the number of unique items. And, for example, $Q(.5)$ corresponds to the 50^{th} precentile.

A.6 Figure creation

The test outputs will be in the form of txt files output to the current working directory. To create the graphs, we have provided scripts that extract the data from these files. The following scripts correspond to the following figures:

- Figure 1 – parseFigure1.py
- Figure 5 – parseAccuracy.py
- All other figures – parseThroughput.py

To use the figures, pass the txt output files to the corresponding script.

A.7 Experiment customization

Each curve in each experiment is customised in the corresponding configure file. The main customisations for the conf files are:

- **Trials_lgMinU / Trials_lgMaxU:** Range of number of unique numbers over which to run the test.
- **LgK:** Log size of the global sketch.
- **CONCURRENT_THETA_localLgK:** Log size of the local sketch.
- **CONCURRENT_THETA_maxConcurrencyError:** Maximum error due to concurrency. For non-eager tests, set to 1.
- **CONCURRENT_THETA_numWriters:** Number of writer threads.
- **CONCURRENT_THETA_numReaders:** Number of background reader threads. For our mixed workload, we used 10 reader threads.
- **CONCURRENT_THETA_ThreadSafe:** Is true if the test should use the concurrent implementation, false if the test should use a lock-based implementation.

A.8 Working with source files

Alternatively, follow the build instructions on Apache DataSketches (Incubating) apache page (https://datasketches.apache.org/), in order to building the above mentioned JAR files, now called:

- incubator-datasketches-java (https://github.com/apache/incubator-datasketches-java)
- incubator-datasketches-memory (https://github.com/apache/incubator-datasketches-memory)
- incubator-datasketches-characterization (https://github.com/apache/incubator-datasketches-characterization)

The version number of incubator-datasketches-java and incubator-datasketches-memory must comply with the version numbers required by incubator-datasketches-characterization. The characterization JAR file is an unsupported open-source code base, and does not pretend to have the same level of quality as the primary repositories. These characterization tests are often long running (some can run for days) and very resource intensive, which makes them unsuitable for including in unit tests. The code in this repository are some of the test suites we use to create some of the plots on our website and provide evidence for our speed and accuracy claims. Alternatively, the datasketches-memory and datasketches-java releases are provided from Maven Central using the Nexus Repository Manager. Go to repository.apache.org and search for "datasketches".

For convenience we have included these repositories as modules in our main repository along with specific branches and commit id's that are known to compile. To compile the jar files:

```
$ git clone https://github.com/ArikRinberg/FastConcurrentDataSketchesArtifact
$ cd FastConcurrentDataSketchesArtifact
$ source customCompile.sh
```

The shell script takes care of initialising the submodules, building the source files, and copying the correct JAR files to the current directory.

Workflow for custom JAR files.

1. After cloning the repository:

```
$ cd FastConcurrentDataSketchesArtifact
```

   In the current working directory, there should be the following JAR files:
   - datasketches-memory-1.1.0-incubating.jar
   - datasketches-java-1.1.0-incubating.jar
   - datasketches-characterization-1.0.0-incubating-SNAPSHOT.jar
2. For each .conf file in the conf_files folder, the following line must be altered:
   From: JobProfile=
   com.yahoo.sketches.characterization.uniquecount.TEST
   To: JobProfile=
   org.apache.datasketches.characterization.theta.concurrent.TEST
   Where TEST is either ConcurrentThetaAccuracyProfile or ConcurrentThetaMultithreadedSpeedProfile.
3. Finally, the following line must be altered in run_test.py:
   From: CMD=
   ‘java -cp "./*" com.yahoo.sketches.characterization.Job ’
   To: CMD=‘java -cp "./*" org.apache.datasketches.Job ’
4. The tests can now be run as explained in Item 3.