Fast Concurrent Data Sketches

Anonymous Author(s)

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 [14], 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, 11], graph mining [13], anomaly (e.g., intrusion) detection [25], real-time data analytics [17], and online classification [23].

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 [14].

Practical sketch implementations have recently emerged in toolkits [1] and data analytics platforms (e.g., PowerDrill [20], Druid [17], Hillview [4], and Presto [7]). 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 [2, 6].

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.

PPoPP'20, 2020, San Diego, CA, USA
2020.

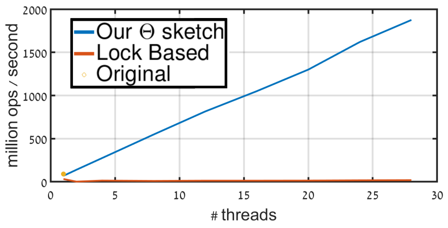


Figure 1. Scalability of DataSketches' Θ sketch protected by a lock vs. our concurrent implementation.

We capitalise on the well-known sketch *mergeability* property [14], 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., [15, 20]), 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 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 [1]: (1) a KMV Θ sketch [11], 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 [3]. 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* due to Henzinger et al. [19] 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 deterministic data structures like stacks [19] and priority queues [10, 22], 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 wrt 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* [18] wrt 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 an important practical problem, 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; we are not aware of any previous application of relaxed consistency to randomised statistical algorithms. 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 studies the scalability of our approach and Section 8 empirically studies the Θ sketch's performance and error with different stream sizes. Finally, Section 9 concludes. The supplementary material 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 explicit 4 definition of *atomic* variables as in Java [5] and C++ [12]. 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. [18] 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 [11] 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 supplementary material Section B.1, 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 $1/\sqrt{k-2}$ [11].

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

We next relax sketch semantics to require that a query return a "valid" value reflecting some subset of the updates that have already been processed but not necessarily all of them. We adopt a variant of Henzinger et al.'s [19] *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 [19] 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.

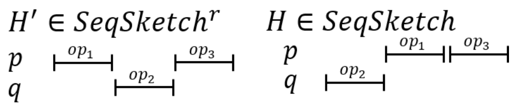


Figure 2. H is a 1-relaxation of H'.

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 parllelises 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 supplementary material 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 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 [24]. 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 ∅                         ▷ 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 supplementary material Section A.2 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 wrt 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

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 $t_i$ , $0 \leq i \leq N$
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:             $globalS.merge(localS_i[1\text{-}cur_i])$
114:             $localS_i[1\text{-}cur_i]$←empty sketch
115:             $prop_i \leftarrow globalS.calcHint()$
116: procedure QUERY(arg)
117:     $localCopy \leftarrow globalS.snapshot(localCopy)$
118:     return $localCopy.query(arg)$
119: procedure UPDATE$_i(a)$
120:     if ¬shouldAdd($hint_i$, $a$) then return
121:     $counter_i \leftarrow counter_i + 1$
122:     $localS_i[cur_i].update(a)$
123:     if $counter_i = b$ then
124:         $prop_i \leftarrow 0$                    ▷ In non-optimised version
125:         wait until $prop_i \neq 0$
126:         $cur_i \leftarrow 1 - cur_i$
127:         $hint_i \leftarrow prop_i$
128:         $counter_i \leftarrow 0$
129:         $prop_i \leftarrow 0$                    ▷ In optimised version
```

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 supplementary material Section A.3 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* [21]. We use forward simulation (with no prophecy variables), ensuring strong linearisability. We conclude the following theorem:

Theorem 1. OptParSketch *instantiated with SeqSketch is strongly linearisable wrt* 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 supplementary material Section B 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 [16]. 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 [11], 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 supplementary material Section B.1. 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 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. Claim 1 in the supplementary material Section B.2 shows that the largest error is always obtained either for $j = 0$ or for $j = r$.

Given an adversary $\mathcal{A}$ that induces an approximation $e_{\mathcal{A}}$, Lemma 9 in the supplementary material Section B.2 proves the following bound:

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

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:

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

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

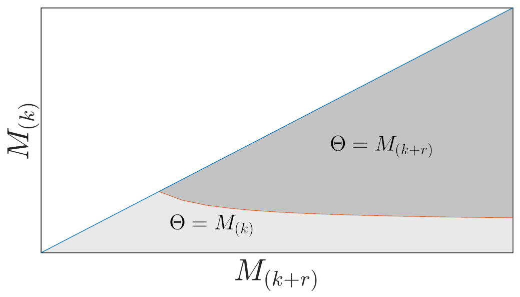


Figure 3. Areas of $M_{(k)}$ and $M_{(k+r)}$. In the dark gray $\mathcal{A}_s$ induces $\Theta = M_{(k+r)}$, and in the light gray, $\Theta = M_{(k)}$. The white area is not feasible.

	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 $r = 8, k = 2^{10}, n = 2^{15}$.

function from order statistics. Equations 3 and 4 in the supplementary material Section B.2 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: $E[n - est(M^r_{(k)})] = E[n - est(M_{(k+j)})] = n - n\frac{k-1}{k+j-1}$. 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 Equation 6 in the supplementary material Section B.2 we show that the RSE bound of Lemma 9 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.

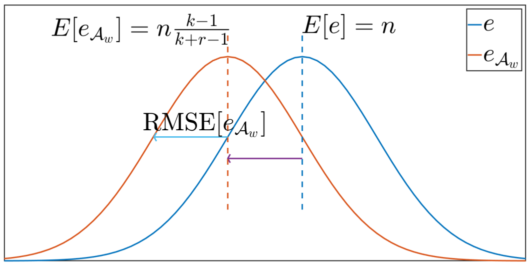


Figure 4. Distribution of estimators e and $e_{\mathcal{A}_w}$. The RSE of $e_{\mathcal{A}_w}$ wrt n is bounded by the relative bias plus the RMSE of $e_{\mathcal{A}_w}$.

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 the original stream is in the range:

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

This can be rewritten as:

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

In Lemma 10 in the supplementary material Section B 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 Scalability evaluation

The purpose of this section is to quantify the scalability of our framework. In this context, the adaptation to small streams is irrelevant so we disable it. Our implementation extends the code in Apache DataSketches (Incubating) [1], a Java open-source library of stochastic streaming algorithms. The Θ sketch implementation 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 largst k values are discarded.

We experimented on a dedicated machine with four Intel Xeon E5-4650 processors, each with 8 cores, for a total of 32 threads (with hyper-threading disabled). We ran an *update-only* workload in which a sketch is built from a very large stream. Repeated experiments showed similar results.

Our baselines are the sequential sketches from the library, wrapped with a read/write lock to allow concurrency. Our results show that adding the lock reduces the single-thread performance of the Θ sketch from 90 to 32 million operations per second, and that of the Quantiles sketch from 77 to 32 million.

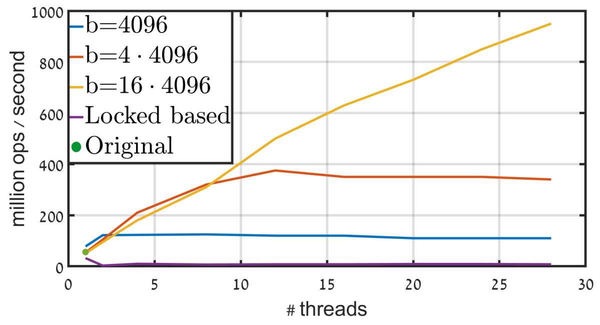


Figure 5. Scalability of DataSketches' Quantiles sketch protected by a lock vs. our concurrent implementation with different local buffer sizes b.

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 $b = 16$ 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. We also tried smaller local buffers, as small as $b = 1$, and the results were similar; for large streams Θ sketch is not sensitive to the buffer size thanks to using shouldAdd to prune out updates. Θ quickly becomes small enough to allow filtering out most of the updates and so even small local buffers fill up slowly.

Our concurrent Quantiles' snapshot method uses a double collect on a dedicated *bitmap* variable in the sketch library. This sketch does not use hints: calcHint returns 1 and shouldAdd always returns true.

In Figure 5 we compare the throughput of our Quantiles sketch to the baseline. We see that again the baseline achieves the best result with a single worker thread. Our sketch, in contrast, scales perfectly when every thread has a sufficiently large local buffer. But here, in order for the background thread to support more worker threads, we have to enlarge the local buffers. This is because, unlike Θ, we do not use a hint to reduce the frequency of updates to local buffers.

8 Evaluating Impact of Stream Size on Theta Sketch

We continue with the Theta sketch experimentation to demonstrate the impact of the stream size on accuracy and performance. In these experiments the hardware is a 12-core Intel Xeon E5-2620 machine. Experiment set-up is described in Section 8.1. We then discuss accuracy and performance results. Finally, Section 8.3 presents tradeoffs between accuracy and performance, which demonstrate the benefits of our stream size-adaptive algorithm.

8.1 Setup

Implementation Concurrent theta sketch is implemented in Java as part of the Apache DataSketches (incubating) project, and is generally available since V0.13.0. The sequential implementation and the sketch at the core of the global sketch in the concurrent implementation are the same (HeapQuickSelectSketch, which is the default sketch family).

Global sketch defines *exact limit* – the limit between exact mode (eager propagation) and estimate mode (lazy propagation). The limit is a function of a configurable error parameter e; the function used in the library is $2/e^2$. Local sketch defines b to be a function of k, e and the maximum number of local threads N. b has positive correlation with e and inverse correlation with N.

Eager propagation, as described in the pseudo-code, requires context switch which incurs high overhead. In the implementation, either the local thread itself eagerly executes every update to the global sketch (buffer size is 1) or lazily delegates updates to a background propagation thread (buffer size is b). The decision is based on the global sketch mode (exact or estimate), which is cached in the local sketch as part of the piggyback process that also updates local theta. As long as the global sketch is in exact mode the propagation critical section is protected by a shared boolean flag indicating whether (eager) propagation is in progress. When the global sketch switches to estimate mode it is guaranteed that no eager propagation gets through; instead local threads attempting to update the global sketch via eager propagation get an indication to pass the buffer via lazy propagation. This implementation ensures (a) local threads avoid costly context switch when the sketch is small, (b) lazy propagation by a background thread is done without synchronization.

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) between 1-5 to accommodate the error bound.

Workload We focus on two simple workloads: (1) write-only - updating a sketch with a stream of unique ids; (2) mixed read-write workload - updating a sketch with background readers querying the number of unique ids in the stream. Background reads refer to threads that occasionally (with 1 ms pauses) query the sketch. This simulates real world scenario where updates are constantly streaming from a feed or multiple feeds, while queries arrive at lower rate.

In both write-only and read-write workloads we measure only put throughput. We vary the number of threads used to feed the sketch from 1 to 12. In all read-write benchmarks we exercise 10 background reader threads.

Framework 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 uniques. 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 avoid measurements 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 uniques per trial) the number of trials is 16.

Accuracy of concurrent theta sketch is measured only in a single-thread environment. As in the performance evaluations the x-axis represents the number of uniques fed into a 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 multiple times (specifically 4K times), logging all estimation and RE results. The y-axis are the lines of the mean and some constant quantiles of the distributions of error measured at each x-axis point on the graph, including the median. This type of graph is called “pitchfork”.

8.2 Results

Accuracy Results The accuracy result of a concurrent theta sketch that does not employ eager propagation are presented in Figure 6a. There are two interesting phenomena to observe. First, it is interesting to see empirical evaluation reflecting the theoretical analysis presented in Section 6.1, and thus the “pitchfork” is distorted toward lower estimation. More specifically, mean relative error is smaller than 0 and values of relative error in all measured quantiles tend to be smaller than the relative error of a sequential implementation (estimating less than a sequential implementation). Thereby increasing or decreasing the absolute error depending on whether the error is negative or positive.

Second, for small streams, when the number of unique items is lower than $2k$, $\theta = 1$ and the estimation is simply the number of items propagated to the global sketch. If eager propagation is not applied, the number of items in the global sketch depends on the delay in propagation. The smaller the sketch is the more significant the delay is, and the mean error reaches as high as 94% (we capped the error in the figure at 10%). As the size of the sketch becomes closer to $2k$ the delay in propagation (of last 16 updates) is less significant, and the mean error decreases. Obviously, most analytic systems today cannot tolerate such high error. For this the algorithm applies eager propagation until the global sketch reaches the desired exact limit. Figure 6b depicts the accuracy results when applying eager propagation. Indeed, the error at small streams (smaller than 8k unique items) is less than 4%.

Write-only Throughput Figure 7a presents throughput measurements of write-only workload. The results are shown in loglog scale. Figure 7b zooms-in on the throughput of large sketches.

When considering large sketches, the concurrent implementation scales with the number of threads; peaking at almost 300M ops/sec with 12 threads. The performance of lock-based implementation, on the other hand, degrades as the contention on the lock increases with the number of threads. Its peak performance is at 25M ops/sec with a single thread. Namely, concurrent theta sketch outperforms lock-based implementation by 12x, and when comparing the performance of 12 threads, concurrent implementation outperforms lock-based implementation by more than 45x.

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

Mixed Workload Figure 8 presents throughput measurements of mixed read-write workload. We compare runs with a single updating thread and 2 updating threads (and 10 background reader threads). Here we see similar trends as in the write-only workload. However, the affect of background readers is more pronounced in lock-based implementation than in concurrent theta sketch. The peak throughput of a single writer-thread in the concurrent implementation is 55M ops/sec with and without background readers. The peak throughput of a single writer thread in the lock-based implementation degrades from 25M to 23M ops/sec; almost 10% slowdown in performance. And recall that this is when readers are not very frequent.

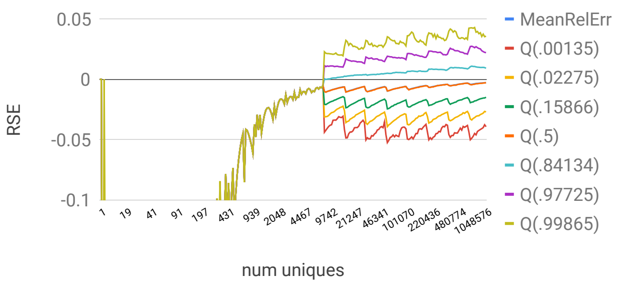
8.3 Accuracy-Throughput Tradeoffs

Eager vs no-eager speedup is presented in Figure 9. It demonstrate the speedup of eager propagation over no-eager propagation for small streams, in addition to the accuracy benefit reported in Figure 6. The improvement goes up to 84x throughput for tiny sketches, and decreases as the sketch grows. The slowdown in performance when the sketch size exceed $2k$ can be explained by the reduction in local buffer size (from $b = 16$ to $b = 5$) in order to accommodate for the required error bound.

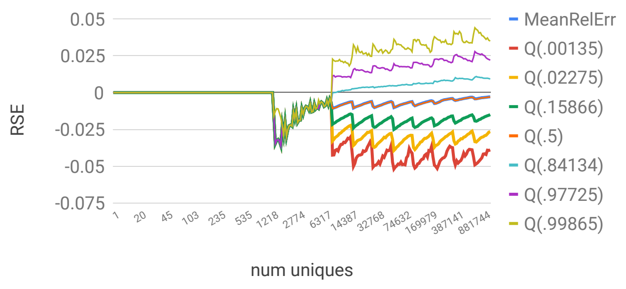
Next we discuss the impact of k. One way to increase the throughput of the concurrent theta sketch is by increasing the size of the global sketch, namely increasing k. On the other hand, this change also increases the error of the estimate. Table 2 presents the tradeoffs between performance and accuracy. Specifically, it presents the crossing-point, the size of the sketch where concurrent implementation outperforms lock-based implementation (both running a single thread), the maximum values (across sketch sizes) of the median error, and 99th percentile error for a variety of k values.

9 Conclusions

Sketches are widely used by a range of applications to process massive data streams and answer queries about them.

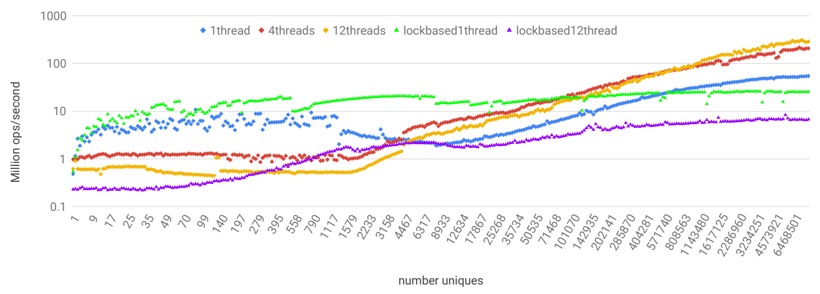


(a) No eager propagation ($e = 1.0$)

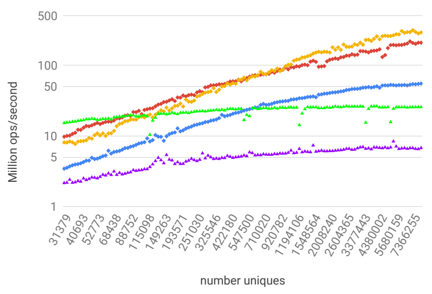


(b) With eager propagation, limit defined by $e = 0.04$

Figure 6. Concurrent Theta Measured Quantiles vs RSE, $k = 4096$.



(a) Throughput, loglog scale



(b) Zooming-in on large sketches

Figure 7. Write-only workload, $k = 4096$, $e = 0.04$.

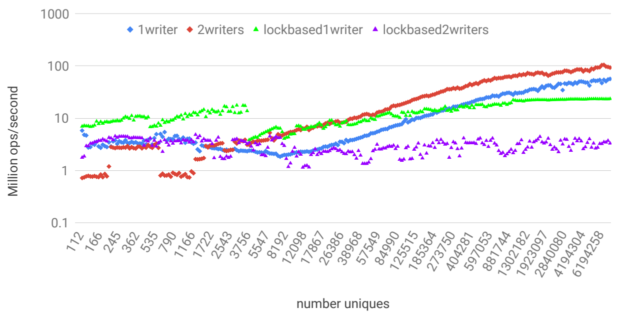


Figure 8. Mixed workloads: writers with background reads, $k = 4096$, $e = 0.04$.

	thpt crossing point	error $Q = 0.5$	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

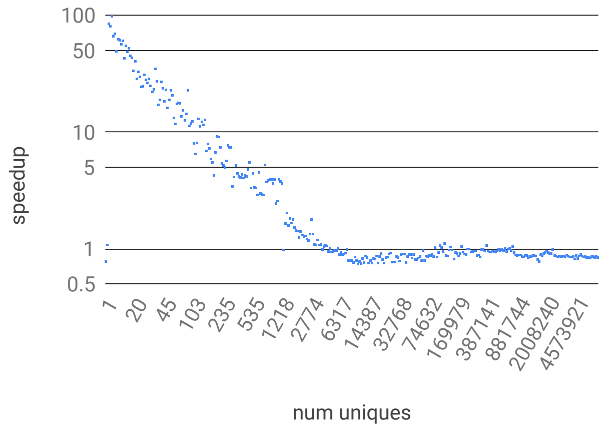


Figure 9. Throughput speedup of eager ($e = 0.04$) vs no-eager ($e = 1.0$) propagation, $k = 4096$.

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 wrt 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

[5] 2011. Java Language Specification: Chapter 17 - Threads and Locks. https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html.

[7] 2018. HyperLogLog in Presto: A significantly faster way to handle cardinality estimation. https://code.fb.com/data-infrastructure/hyperloglog/.

[1] 2019. Apache DataSketches (Incubating). https://incubator.apache.org/clutch/datasketches.html.

[2] 2019. ArrayIndexOutOfBoundsException during serialization. https://github.com/DataSketches/sketches-core/issues/178#issuecomment-365673204.

[3] 2019. DataSketches: Concurrent Theta Sketch Implementation. https://github.com/apache/incubator-datasketches-java/blob/master/src/main/java/com/yahoo/sketches/theta/ConcurrentDirectQuickSelectSketch.java.

[4] 2019. Hillview: A Big Data Spreadsheet. https://research.vmware.com/projects/hillview.

[6] 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, 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

[11] 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.

[12] 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

[13] 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

[14] Graham Cormode. 2017. Data Sketching. *Queue* 15, 2, Article 60 (April 2017), 19 pages. https://doi.org/10.1145/3084693.3104030

[15] Graham Cormode, S Muthukrishnan, and Ke Yi. 2011. Algorithms for distributed functional monitoring. *ACM Transactions on Algorithms (TALG)* 7, 2 (2011), 21.

[16] Herbert Aron David and Haikady Navada Nagaraja. 2004. Order statistics. *Encyclopedia of Statistical Sciences* 9 (2004).

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Nancy A Lynch. 1996. *Distributed algorithms*. Elsevier.

[22] Hamza Rihani, Peter Sanders, and Roman Dementiev. 2014. Multi-queues: Simpler, faster, and better relaxed concurrent priority queues. *arXiv preprint arXiv:1411.1209* (2014).

[23] 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

[24] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)* 11, 1 (1985), 37–57.

[25] 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 Proofs

In Section A.1 we introduce some formalisms. In Section A.2 we prove that the unoptimised algorithm is strongly linearisable with respect to the relaxed specification $SeqSketch^r$ with $r = Nb$. Finally, in Section A.3 we show that the the optimised algorithm is strongly linearisable with respect to the relaxed specification $SeqSketch^r$ with $r = 2Nb$.

A.1 Definitions

Note that the only methods invoked by $ParSketch$ on $globalS$ are snapshot and merge, and since merge is only invoked by t_0, the only concurrency is between a snapshot and another operation (snapshot or merge). Recall that we required such executions of a composable sketch to be strongly linearisable. By slight abuse of terminology, we refer to these operations as atomic steps, for example, we refer to the linearisation point of $globalS$.merge simply as "$globalS$.merge step".

Likewise, as $localS_i$ is only accessed sequentially by a single thread, either t_i or t_0 (using $prop_i$ to synchronise), we refer to the method calls shouldAdd and update as atomic steps.

Because we prove only safety properties, we restrict out attention to finite executions. For analysis purposes we use abstract counters:

- An array $sig\_ctr[N]$, that counts the number of times each thread t_i signals to the propagator (line 124).
- An array $merge\_ctr[N]$ counting the number of times t_0 executes a merge with thread t_i's local sketch (line 113).

Recall that in Section 3, we said that a sketch *summarises* a stream or a sequential history if its state is the state of a sketch that has processed the stream or history. We now overload the term "summarises" to apply also to threads.

Definition A.1 (Thread summary)**.** Consider a time t in an execution σ of Algorithm 2. If at time t either $prop_i \neq 0$ or $sig\_ctr[i] > merge\_ctr[i]$, then we say that update thread t_i *summarises* the history summarised by $localS_i$ at time t. Otherwise, thread t_i summarises the empty history at time t. The propagator thread t_0 summarises the same history as $globalS$ at any time during an execution σ.

As we want to analyse each thread's steps in an execution, we first define the projection from execution σ onto a thread t_i.

Definition A.2 (Projection)**.** Given a finite execution σ and a thread t_i, $\sigma\Big|_{t_i}$ is the subsequence of σ consisting of steps taken by t_i.

We want to prove that each thread's summary corresponds to the sequence of updates processed by that thread since the last propagation, taking into account only those that alter local state variables. These are updates for which *shouldAdd* returns true.

Definition A.3 (Unprop updates)**.** Given a finite execution σ, we denote by $\mathrm{suff}_i(\sigma)$ the suffix of $\sigma\Big|_{t_i}$ starting at the last $globalS$.merge($localS_i$) event, or the beginning of σ if no such event exists. The unprop suffix $\mathrm{up\_suff}_i(\sigma)$ of update thread i is the subsequence of $\mathcal{H}(\mathrm{suff}_i(\sigma))$ consisting of $update(a)$ executions in $\mathrm{suff}_i(\sigma)$ for which shouldAdd($hint_i$, arg) returns true in line 120.

We define the relation between a sequential history H and a stream A.

Definition A.4. 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 kth update in H.

Finally, we define the notion of *happens before* in a sequential history H.

Definition A.5. Given a finite sequential history H and two method invocations M_1, M_2 in H, we denote $M_1 \prec_H M_2$ if M_1 precedes M_2 in H.

A.2 Unoptimised algorithm proof

Our strong linearisability proof uses two mappings, f and l, from executions to sequential histories defined as follows. For an execution σ of $ParSketch$, we define a mapping f by ordering operations according to *visibility points* defined as follows:

- For a query, the visibility point is the snapshot operation it executes.
- For an $\mathrm{update}_i(a)$ where shouldAdd($prop_i$, a) returns false at time t, its visibility point is t.
- Otherwise, for an $\mathrm{update}_i(a)$, let t be the first time after its invocation in σ when thread i changes $prop_i$ to 0 (line 124). Its visibility point is the (linearisation point of the) first merge that occurs with $localS_i$ after time t. If there is no such time, then $\mathrm{update}_i(a)$ does not have a visibility point, i.e., is not included in $f(\sigma)$

Note that in the latter case, the visibility point may occur after the update returns, and so f does not necessarily preserve real-time order.

We also define a mapping l by ordering operations according to *linearisation points* define as follows:

- An updates' linearisation point is its invocation
- A query's linearisation point is its visibility point.

By definition, $l(\sigma)$ is prefix-preserving.

We show that for every execution σ of $ParSketch$, (1) $f(\sigma) \in SeqSketch$, and (2) $f(\sigma)$ is an r-relaxation of $l(\sigma)$ for $r = Nb$. Together, this implies that $l(\sigma) \in SeqSketch^r$, as needed.

We first show that $Prop_i \neq 0$ if t_i's program counter is not on lines 124 or 125.

Invariant 1. *At any time during a finite execution σ of ParSketch for every $i = 1, \ldots, N$, if t_i's program counter isn't on lines 124 or 125, then $prop_i \neq 0$.*

Proof. The proof is derived immediately from the algorithm: $prop_i$ is initialised to 1 and gets the value of 0 on line 124, and then waits on line 125 until $prop_i \neq 0$. After continuing passed line 125, $prop_i \neq 0$ again. □

We also observe the following:

Observation 1. *Given a finite execution σ of ParSketch, for every $i = 1, \ldots, N$, every execution of $globalS.merge(localS_i)$ in σ (line 113) is preceded by an execution of $prop_i \leftarrow 0$ (line 124).*

We observe the following relationship between t_i's program counter and $sig\_ctr[i]$ and $merge\_ctr[i]$:

Observation 2. *At all times during a finite execution σ of ParSketch, for every $i = 1, \ldots, N$, $merge\_ctr[i] \leq sig\_ctr[i] \leq merge\_ctr[i] + 1$. Moreover, if t_i's program counter isn't on lines 124 or 125, then $sig\_ctr[i] = merge\_ctr[i]$.*

We show that at every point in an execution, update thread t_i summarises up_suff$_i(\sigma)$. In essence, this means that we have not "forgotten" any updates.

Invariant 2. *At all times during a finite execution σ of ParSketch, for every $i = 1, \ldots, N$, t_i summarises up_suff$_i(\sigma)$.*

Proof. The proof is by induction on the length of σ. The base is immediate. Next we consider a step in σ that can alter the invariant. We assume the invariant is correct for σ', and prove correctness for $\sigma = \sigma', step$. We consider ony steps that can alter the invariant, meaning the step can either lead to a change in up_suff$_i(\sigma)$, or a change in the history summarised by t_i. This means we need consider only 4 cases:

- A step $localS_i.update(arg)$ (line 122) by thread t_i.
  In this case, up_suff$_i(\sigma)$ =up_suff$_i(\sigma')$, $update(arg)$. By the inductive hypothesis, before the step $localS_i$ summarises up_suff$_i(\sigma')$, and so after the update, $localS_i$ summarises up_suff$_i(\sigma')$, $update(arg)$ = up_suff$_i(\sigma)$. From Invariant 1 $prop_i \neq 0$, therefore, by Definition A.1, t_i summarises the same history as $localS_i$, i.e., up_suff$_i(\sigma)$, preserving the invariant.
- A step $prop_i \leftarrow 0$ (line 124) by thread t_i.
  By the inductive hypothesis, before the step $localS_i$ and t_i summarise the same history up_suff$_i(\sigma')$. As no update occurs, up_suff$_i(\sigma')$=up_suff$_i(\sigma)$. The step doesn't alter $localS_i$, so after the step, $localS_i$ still summarises up_suff$_i(\sigma)$. On this step the counter $sig\_ctr[i]$ is increased but $merge\_ctr[i]$ is not, so $sig\_ctr[i] > merge\_ctr[i]$. Therefore, by Definition A.1, t_i summarises the same history as $localS_i$, namely up_suff$_i(\sigma)$, preserving the invariant.
- A step $globalS.merge(localS_i)$ (line 113) by thread t_0.
  By Definition A.3, after this step up_suff$_i(\sigma)$ is empty. As this step is a $merge$, $merge\_ctr[i]$ is increased by one, so $sig\_ctr[i] = merge\_ctr[i]$ by Observation 2. Therefore, by Definition A.1, t_i summarises the empty history, preserving the invariant.
- A step $prop_i \leftarrow globalS.calcHint()$ (line 115) by thread t_0
  Before executing the step, t_0 executed line 114. Thread t_i is waiting for $prop_i \neq 0$ on line 125, therefore has not updated $localS_i$. Therefore, $localS_i$ summarises the empty history. As a merge with thread i was executed and no updates have been invoked, up_suff$_i(\sigma)$ is the empty history. The function $calcHint$ cannot return 0, therefore after the step $prop_i \neq 0$. By Definition A.1, t_i summarises the same history as $localS_i$, i.e., the empty history. Therefore, t_i summarises up_suff$_i(\sigma)$, preserving the invariant.

□

Next, we prove that t_0 summarises $f(\sigma)$.

Invariant 3 (History of propagator thread)**.** *Given a finite execution σ of ParSketch, t_0 summarises $f(\sigma)$.*

Proof. The proof is by induction on the length of σ. The base is immediate. We assume the invariant is correct for σ', and prove correctness for $\sigma = \sigma', step$. There are two steps that can alter the invariant.

- A step $globalS.merge(localS_i)$ (line 113) by thread t_0.
  By the inductive hypothesis, before the step, t_0 summarises $f(\sigma')$. And by Invariant 2, before the update, t_i summarises up_suff$_i(\sigma')$, and bu Invariant 1 $localS_i$ summarises the same history. Let $A = \mathcal{S}(f(\sigma))$, and $B = \mathcal{S}(\text{up\_suff}_i(\sigma'))$. After the merge *globalS* summarises $A||B$. Therefore, t_0 summarises $f(\sigma)$ preserving the invariant.
- A step shouldAdd($prop_i$, a) (line 120) by thread t_i, returning false.
  Let H be that last hint returned to t_i, and let σ'' be the prefix of σ up to this point. By the induction hypothesis, at that point *globalS* summarised $f(\sigma'')$. Let $A = \mathcal{S}(f(\sigma''))$, and let $B = \mathcal{S}(f(\sigma'))$, and let B_1 be such that $B = A||B_1$. By the induction hypothesis, before the step, *globalS* summarises $B = A||B_1$. By the assumption of *shouldAdd*, if shouldAdd(H, *arg*) returns false, then if a sketch summarises $B = A||B_1||B_2$, then it also summarises $B = A||B_1||a||B_2$. Let $B_2 = \emptyset$, then *globalS* summarises $B = A||B_1||B_2$, therefore also summarises $A||B_1||a||B_2 = A||B_1||a$. Therefore, after the step, *globalS* summarises $f(\sigma)$ preserving the invariant.

□

To finish the proof that $f(\sigma) \in SeqSketch$, we prove that a query invoked at the end of σ returns a value equal to the value returned by a sequential sketch after processing $A = \mathcal{S}(f(\sigma))$.

Lemma 2 (Query Correctness)**.** *Given a finite execution σ of ParSketch, let Q be a query that returns in σ, and let v be Q's visibility point. Let σ' be the prefix of σ until point v, and let $A = \mathcal{S}(f(\sigma'))$. Q returns a value that is equal to the value returned by a sequential sketch after processing A.*

Proof. Let σ be an execution of *ParSketch*, and let Q be a query that returns in σ. Let σ' and A be as defined in the lemma. By Invariant 3, t_0 summarises $f(\sigma')$ at point v, therefore *globalS* summarises $f(\sigma')$ at the same point, therefore *globalS* summarises stream A at point v. The visibility point for the query, at point v, is *globalS*.snapshot(). By the requirement from *S*.snapshot(), for all *arg* $globalS.query(arg) = localCopy.query(arg)$. Because *globalS* summarises stream *A*, $localCopy.query(arg)$ returns a value equal to the value returned by the sequential sketch *globalS* after processing *A*. □

As we have proven that each query in $f(\sigma)$ returns a value that estimates all the updates that happen before its invocation, we have proven the following:

Lemma 3. *Given a finite execution σ of ParSketch, $f(\sigma) \in SeqSketch$.*

To complete the proof, we prove that $f(\sigma)$ is an r-relaxation of $l(\sigma)$, for $r = Nb$. We begin by proving orders between queries and other method calls.

Lemma 4. *Given a finite execution σ of ParSketch, and given an operation O(query or update) in $l(\sigma)$, for every Q in $l(\sigma)$ such that $Q \prec_{l(\sigma)} O$, then $Q \prec_{f(\sigma)} O$.*

Proof. If O is a query, then proof is immediate from the definitions of l and f. If O is an update, then, by the definition of f, an updates visibility point is at the earliest its linearisation point. As Q's visibility point and linearisation point are equal, it follows that if $Q \prec_{l(\sigma)} O$ then $Q \prec_{f(\sigma)} O$. □

We next prove an upper bound on the number of updates in up_suff$_i(\sigma)$. We denote the number of updates in history H as $|H|$.

Lemma 5. *Given a finite execution σ of ParSketch, $|up\_suff_i(\sigma)| \le b$.*

Proof. As $counter_i$ is incremented before an update which is included in up_suff$_i(\sigma)$, it follows that $|\text{up\_suff}_i(\sigma)| \le counter_i$. When $counter_i = b$, t_i signals for a propagation (line 124) and then waits until $prop_i \neq 0$ (line 125). When t_i finishes waiting, then it zeros the counter (line 128) before ingesting more updates, therefore, $count_i \le b$. Therefore, it follows that $|\text{up\_suff}_i(\sigma)| \le b$. □

As $f(\sigma)$ contains all updates with visibility points, we can now prove the following.

Lemma 6. *Given a finite execution σ of ParSketch, $|f(\sigma)| \ge |l(\sigma)| - Nb$.*

Proof. From Lemma 5, $|\text{up\_suff}_i(\sigma)| \le b$. The only updates without a visibility point are updates that are in $\text{up\_suff}_i(\sigma)$ for some i. Therefore $f(\sigma)$ contains all updates but any update in a history $\text{up\_suff}_i(\sigma)$ for some i. There are N update threads, therefore $|f(\sigma)| = |l(\sigma)| - \sum_{i=1}^{N} |\text{up\_suff}_i(\sigma)|$ so $|f(\sigma)| \ge |l(\sigma)| - Nb$. □

We will now prove that given an execution σ of *ParSketch*, every invocation in $f(\sigma)$ is preceded by all but at most r of the invocations in $l(\sigma)$.

Lemma 7. *Given a finite execution σ of ParSketch, $f(\sigma)$ is a r-relaxation of $l(\sigma)$.*

Proof. Let σ be a finite execution of *ParSketch*, and consider an operation O in $f(\sigma)$ such that O is also in $l(\sigma)$. Let $Ops = \{ O' \mid (O' \prec_{l(\sigma)} O) \wedge (O' \not\prec_{f(\sigma)} O) \}$. We show that $|Ops| \le r$. By Lemma 4, for every query Q in $l(\sigma)$ such that $Q \prec_{l(\sigma)} O$, then $Q \prec_{f(\sigma)} O$, meaning $Q \notin Ops$. Let σ^{pre} be a prefix and σ^{post} a suffix of σ such that $l(\sigma) = l(\sigma^{pre}), O, l(\sigma^{post})$. From Lemma 6, $|f(\sigma^{pre})| \ge |l(\sigma^{pre})| - r$. As $|f(\sigma^{pre})|$ is the number of updates in $f(\sigma^{pre})$, and $|l(\sigma^{pre})|$ is the number of updates in $l(\sigma^{pre})$, $f(\sigma^{pre})$ contains all but at most r updates in $l(\sigma^{pre})$. As $l(\sigma^{pre})$ contains all the updates that precede O. Meaning Ops is all the updates in $l(\sigma^{pre})$ and not in $f(\sigma^{pre})$. Therefore, $|Ops| = |l(\sigma^{pre})| - |f(\sigma^{pre})| \le r$. Therefore, by Definition 4.1, $f(\sigma)$ is an r-relaxation of $l(\sigma)$. □

Putting together Lemma 3 and Lemma 7, we have shown that given a finite execution σ of *ParSketch*, $f(\sigma) \in SeqSketch$ and $f(\sigma)$ is an r-relaxation of $l(\sigma)$. We have proven Lemma 1.

A.3 Optimised algorithm proof

We prove the correctness of the optimised algorithm by simulating the optimised version of Algorithm 2 with the unoptimised version. We denote the optimised version of Algorithm 2 as *OptParSketch*. We show a *simulation relation* [21], thus proving that *OptParSketch* is strongly linearisable with regards to $SeqSketch^{2Nb}$.

Consider an arbitrary worker thread t_i for the optimised algorithm, and simulate this thread using two worker threads t_i^0, t_i^1 of Algorithm 2. To simulate N worker threads, you need $2N$ threads, and they are mapped the same way.

The idea behind the simulation is that there is a delay in when the *hint* returned to the worker thread is used for pre processing, so we can simulate each thread by two thread. For example in Figure 10, each block A_i is a stream such that b updates pass the test of *shouldAdd* (except maybe A_n). The stream processed by t_i is $A = A_1||A_2||\ldots||A_n$ and we assume n is even. Each A_i is evaluated against the *hint* written above it. The thread t_i^0 simulates processing $A_1||A_3||\ldots||A_{n-1}$, and thread t_i^1 simulates processing $A_2||A_4||\ldots||A_n$.

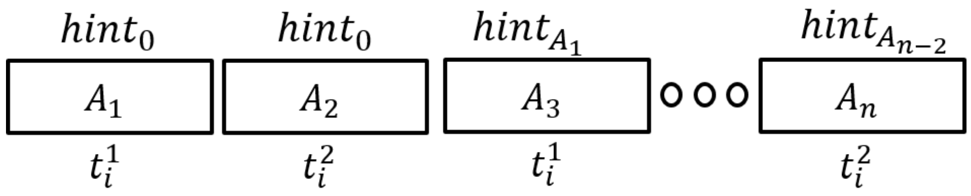


Figure 10. Simulation of processing $A = A_1||A_2||\ldots||A_n$.

The simulation uses abstract variables oldHint_i^1, and oldHint_i^1, both initialised to 1. These variables are updated with the flipping of cur_i (line 126), such that:

- oldHint_i^1 is updated with the current (pre-flip) value of $hint_i$
- oldHint_i^2 is updated with the current (pre-flip) value of $oldHint_i^0$

In addition, the simulation uses an abstract variable $auxCount_i$ initialised to 0. This variable is set to b before the first execution of line 126, and is never changed after that.

Finally, the simulation uses two abstract variables PC_i^0 and PC_i^1 to be program counters for threads t_i^0 and t_i^1. They are initialised to *Idle*.

For simplicity, we use an array of size 2 of threads t, such that $t[0] = t_i^0$ and $t[1] = t_i^1$.

We define a mapping g from the state of *OptParSketch* to the state of *ParSketch* as follows:

- *globalS* in *OptParSketch* is mapped to *globalS* in *ParSketch*.
- $\text{localS}_i[j]$ is mapped to $t[j]$.localS for $j = 0, 1$.
- counter_i is mapped to $t[cur_i]$.counter.

- auxCount is mapped to $t[1 - cur_i]$.counter.
- hint$_i$ is mapped to $t[cur_i]$.hint and $t[cur_i]$.prop if t_i is not right before executing line 127, otherwise oldHint$_i^2$ is mapped to $t[cur_i]$.hint and prop$_i$ is mapped to $t[cur_i]$.prop.
- prop$_i$ is mapped to $t[1 - cur_i]$.prop if t_i is not right before executing lines 127-129, otherwise oldHint$_i^1$ is mapped to $t[1 - cur_i]$.prop.
- oldHint$_i^1$ is mapped to $t[1 - cur_i]$.hint.

For example, Figure 11 shows a mapping when cur_i equals 0, before executing line 127. Table 3 shows the steps taken by t_i^0 and t_i^1 when $cur_i = 0$ before line 123.

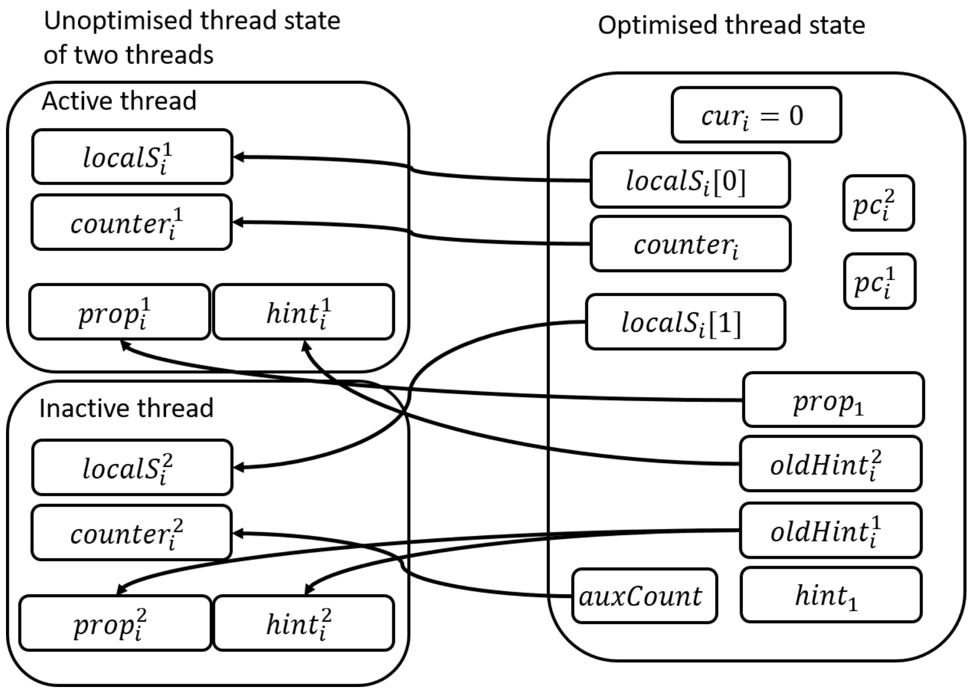


Figure 11. Reference mapping of g when cur_i equals 0 before executing lines 127.

OptParSketch line	*ParSketch* line	Executing thread
123	123	t_i^0
125	125	t_i^1
126	-	-
127	127	t_i^1
128	128	t_i^1
129	124	t_i^0

Table 3. Example for steps taken by t_i^0 and t_i^1 for each step taken by t_i when $cur_i = 0$ before line 123, meaning the "round" of b updates was ingested by t_i^0. On line 126 neither thread takes a step.

We also define the steps taken in *ParSketch* when *OptParSketch* takes a step. If a *query* is invoked, then both algorithms take the same step. If an *update* in invoked, the an *update* is invoked in $t[cur_i]$ in *ParSketch*. If the counter gets up to b (meaning we get to line 125), then $t[1 - cur_i]$ executes line 125. When *OptParSketch* flips cur_i (line 126), then neither threads t_i^0 or t_i^1 take a step. Afterwards, lines 127 and 128 execute the respective lines on thread $t[cur_i]$, and line 129 executes 124 on thread $t[1 - cur_i]$.

Lemma 8. *g is a simulation relation from OptParSketch to ParSketch.*

Proof. The proof is by induction on the steps in an execution. In the initial state, the mapping trivially holds. In a given step, we refer to $t[cur_i]$ as the *active* thread and $t[1 - cur_i]$ as the *inactive thread.* Query threads trivially map to themselves and do not alter the state. We next consider update and propagator threads. Consider first the steps of OptParSketch that execute the corresponding step on the active thread. These are lines 119-123 and 127-128. All of these steps have a direct one-to-one correspondence with the same steps of ParSketch in the active thread ($t[cur_i]$), and except in line 127 abd 129, the affected state variables are mapped to the same state variables in the active thread. So these steps trivially preserve g. Line 124 in *ParSketch* is executed on the inactive thread when *OptParSketch* executes line 129. As after this step the inactive thread's prop and prop_i are both 0, g is preserved. Line 125 is executed on the inactive thread, waiting on the same variable, and modifies no variables, so g is preserved.

Line 126 flips cur_i and neither thread takes a step in *ParSketch*. Here, the mappings of *prop*, *hint*, and *counter* change. On this step oldHint_i^1 and oldHint_i^2 are updated as defined, and as t_i is right before executing line 127, oldHint_i^1 is equal to the inactive thread's ($t[1 - cur_i]$) hint, and as before the step the (now) inactive thread's prop was equal to hint_i, then after this step it is equal to oldHint_i^0. As before the step the (now) active thread's hint was equal to oldHint_i^1, after this step it is equal to oldHint_i^2. Finally, as before the step the (now) active thread's prop was equal to prop_i, after this step it remains equal to prop_i, so this step preserve g.

In line 127, hint_i gets the value of prop_i, and the same happens on the active thread. As before this line the active thread's prop was equal to prop_i, after this step the inactive thread's prop and hint are equal to hint_i, preserving g. As the active thread's counter is equal to $counter_i$, line 128 preserves g. The now inactive thread has filled its local sketch, therefore its counter is b, which equals auxCount. Finally, the propagator thread's steps (lines 110-115) execute on the inactive thread and it is easy to see that all variables accessed in these steps are mapped to the same variables in the inactive thread. □

Note that the simulation relation uses no prophecy variables, i.e., does not "look into the future". Thus, the mapping of all ParSketch's steps, including linearisation points, to steps in OptParSketch is prefix-preserving. Since we use two update threads of ParSketch to simulate one thread in OptParSketch, we have proven the following Theorem:

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

B Derivations of error bounds

B.1 RSE and RSME definition

Given n i.i.d random variable $X_1, \ldots, X_n$, the *Relative Standard Error (RSE)* of some estimate $E = f(X_1, \ldots, X_n)$ for some function f with a PDF $f_E(e)$ from some value $\hat{e}$, is defined to be:

$$\text{RSE}[E] = \sqrt{\int_{-\infty}^{\infty} \frac{(e - \hat{e})^2}{n^2} f_E(e) de}$$

Furthermore, the *Relative Standard Mean Error (RSME)* is the RSE for $\hat{e} = E[E]$, i.e., the relative standard error from the mean. The RSME can be expanded to:

$$\text{RSME}[E] = \sqrt{\int_{-\infty}^{\infty} \frac{(e - \hat{e})^2}{n^2} f_E(e) de} = \sqrt{\frac{\sigma^2(E)}{n}}$$

B.2 Θ sketch error bounds

Lemma 9. *The RSE of $e_{\mathcal{A}}$ satisfies the inequality $RSE[e_{\mathcal{A}}] \le \sqrt{\frac{\sigma^2(e_{\mathcal{A}})}{n^2}} + \sqrt{\frac{(E[e_{\mathcal{A}}]-n)^2}{n^2}}$.*

Proof. Using the definition given in the supplementary material Section B.1:

$$\begin{aligned}
\mathrm{RSE}[e_{\mathcal{A}}]^2 &= \frac{1}{n^2}\int_{-\infty}^{\infty}(e-n)^2\cdot f_{e_{\mathcal{A}}}(e)\,de \\
&= \frac{1}{n^2}\int_{-\infty}^{\infty}(e-E[e_{\mathcal{A}}]+E[e_{\mathcal{A}}]-n)^2\cdot f_{e_{\mathcal{A}}}(e)\,de \\
&\le \frac{1}{n^2}\int_{-\infty}^{\infty}\left((e-E[e_{\mathcal{A}}])^2+(E[e_{\mathcal{A}}]-n)^2\right)\cdot f_{e_{\mathcal{A}}}(e)\,de \\
&= \frac{\sigma^2(e_{\mathcal{A}})+(E[e_{\mathcal{A}}]-n)^2}{n^2} \\
\mathrm{RSE}[e_{\mathcal{A}}] &\le \sqrt{\frac{\sigma^2(e_{\mathcal{A}})}{n^2}}+\sqrt{\frac{(E[e_{\mathcal{A}}]-n)^2}{n^2}}
\end{aligned}$$

□

Claim 1. *Consider j values X_i, $1 \le i \le j$, in the interval $[0,1]$, let $M_{(i)}$ be the i^{th} minimum value among the j. The X_i that maximises $|\frac{k-1}{x}-n|$ for a given n is either $M_{(0)}$ or $M_{(j)}$.*

Proof. Assume for the sake of contradiction that the variable that maximises $|\frac{k-1}{x}-n|$ is $M_{(i)}$ for $0 < i < j$. We consider two cases:

- If $\frac{k-1}{M_{(i)}} \le n$, as $M_{(j)} > M_{(i)}$ then $\frac{k-1}{M_{(j)}} < \frac{k-1}{M_{(i)}} \le n$, therefore $|\frac{k-1}{M_{(j)}}-n| > |\frac{k-1}{M_{(i)}}-n|$, which is a contradiction.
- If $\frac{k-1}{M_{(i)}} > n$, as $M_{(0)} < M_{(i)}$ then $\frac{k-1}{M_{(0)}} > \frac{k-1}{M_{(i)}} > n$, therefore $|\frac{k-1}{M_{(0)}}-n| > |\frac{k-1}{M_{(i)}}-n|$, which is a contradiction.

□

Computation of $e_{\mathcal{A}_s}$ expectation and error Recall the the $\mathcal{A}_s$ knows the oracles coin flips, therefore knows $M_{(k)}$ and $M_{(k+r)}$, and chooses $M^r_{(k)}$ accordingly. Therefore, we our analysis is on the order statistics of the full stream, as it is this that the adversary sees. From order statistics, the joint probability density function of $M_{(k)}, M_{(k+r)}$ is:

$$f_{M_{(k)},M_{(k+r)}}(m_k, m_{k+r}) = n!\frac{m_k^{k-1}}{(k-1)!}\frac{(m_{k+r}-m_k)^{r-1}}{(r-1)!}\frac{(1-m_{k+r})^{n-(k+r)}}{(n-(k+r))!}.$$

The expectation of $e_{\mathcal{A}_s}$ and $e^2_{\mathcal{A}_s}$ can be computed as follows:

$$\begin{aligned}
E[e_{\mathcal{A}_s}] &= \int_0^1\int_0^{m_k} e_{\mathcal{A}_s}\cdot f_{M_{(k)},M_{(k+r)}}(m_k,m_{k+r})\,dm_{k+r}\,dm_k \\
E[e^2_{\mathcal{A}_s}] &= \int_0^1\int_0^{m_k} \left[e_{\mathcal{A}_s}\right]^2\cdot f_{M_{(k)},M_{(k+r)}}(m_k,m_{k+r})\,dm_{k+r}\,dm_k
\end{aligned} \tag{3}$$

Finally, the RSE of $e_{\mathcal{A}_s}$ is derived from the standard error of $e_{\mathcal{A}_s}$:

$$\begin{aligned}
\text{RSE}[e_{\mathcal{A}_s}]^2 &= \frac{1}{n^2}\int_0^1\int_0^{m_k}\left(e_{\mathcal{A}_s}-n\right)^2\cdot f_{M_{(k)},M_{(k+r)}}(m_k,m_{k+r})\,dm_{k+r}\,dm_k \\
&= \frac{1}{n^2}\int_0^1\int_0^{m_k}\left(e_{\mathcal{A}_s}-E[e_{\mathcal{A}_s}]+E[e_{\mathcal{A}_s}]-n\right)^2\cdot f_{M_{(k)},M_{(k+r)}}(m_k,m_{k+r})\,dm_{k+r}\,dm_k \\
&\le \frac{1}{n^2}\left(\sigma^2(e_{\mathcal{A}_s})+(e_{\mathcal{A}_s}-n)^2\right) \\
\text{RSE}[e_{\mathcal{A}_s}] &\le \sqrt{\frac{\sigma^2(e_{\mathcal{A}_s})+(e_{\mathcal{A}_s}-n)^2}{n^2}} \\
&\le \sqrt{\frac{\sigma^2(e_{\mathcal{A}_s})}{n^2}}+\sqrt{\frac{(e_{\mathcal{A}_s}-n)^2}{n^2}}
\end{aligned} \tag{4}$$

Computation of $e_{\mathcal{A}_w}$ ***expectation and error*** Recall that $\mathcal{A}_w$ always hides r element smaller than Θ, thus forcing $M^r_{(k)} = M_{(k+r)}$. Here too our analysis is on the order statistics of the full stream, as this is what the adversary sees. The expectation of $e_{\mathcal{A}_w}$ and $e^2_{\mathcal{A}_w}$ is computed using well known equations from order statistics:

$$\begin{aligned}
E[e_{\mathcal{A}_w}] &= E\left[\frac{k-1}{M_{(k+r)}}\right] = n\frac{k-1}{k+r-1} \\
E[e^2_{\mathcal{A}_w}] &= (k-1)^2\frac{n(n-1)}{(k+r-2)(k+r-1)} \\
\sigma^2[e_{\mathcal{A}_w}] &= E[e^2_{\mathcal{A}_w}]-E[e_{\mathcal{A}_w}]^2 \\
&= (k-1)^2\frac{n(n-1)}{(k+r-2)(k+r-1)}-\left(n\frac{k-1}{k+r-1}\right)^2 \\
&< \frac{n(k-1)^2}{k+r-1}\left[\frac{n}{(k+r-2)(k+r-1)}\right] \\
\sigma^2[e_{\mathcal{A}_w}] &< \frac{n^2}{k+r-2}
\end{aligned}$$

We derive following equation:

$$\sqrt{\frac{\sigma^2[e_{\mathcal{A}_w}]}{E[e_{\mathcal{A}_w}]}} < \frac{1}{k-2} \tag{5}$$

Finally, the RSE of $e_{\mathcal{A}_w}$ is derived from the standard error of $e_{\mathcal{A}_w}$, and as $E[e_{\mathcal{A}_w}] < n$, and using the same "trick" as in Equation 4:

$$\begin{aligned}
\text{RSE}[e_{\mathcal{A}_w}]^2 &= \frac{1}{n^2}\int_0^1\left(e_{\mathcal{A}_w}-n\right)^2\cdot f_{M_{(k+r)}}(m_{k+r})\,dm_{k+r} \\
&< \frac{1}{n^2}\left(\sigma^2(e_{\mathcal{A}_w})+(E[e_{\mathcal{A}_w}]-n)^2\right) \\
\text{RSE}[e_{\mathcal{A}_w}] &< \sqrt{\frac{\sigma^2(e_{\mathcal{A}_w})}{E[e_{\mathcal{A}_w}]^2}}+\sqrt{\frac{(E[e_{\mathcal{A}_w}]-n)^2}{n^2}}
\end{aligned}$$

Using Equation 5:

$$\text{RSE}[e_{\mathcal{A}_w}] < \sqrt{\frac{1}{k-2}}+\frac{r}{k-2} \tag{6}$$

B.3 Quantiles sketch error bounds

We begin by analysing the range given in Equation 2 for $0 \le \phi \le 0.5$.

Claim 2. *For $0 \leq \phi \leq 0.5$ and $i, j > 0$ such that $0 \leq i+j \leq r$ and $\epsilon < 0.5$, then: (1) $(1-\phi)i - \phi j + \epsilon(n-(i+j)) \leq (1-\phi)r + \epsilon(n-r)$, and (2) $\phi j - (1-\phi)i + \epsilon(n-(i+j)) \leq (1-\phi)r + \epsilon(n-r)$.*

Proof. As $\phi \leq 0.5$, and $\epsilon \ll 0.5$ then $1-\phi-\epsilon > 0$. As $0 \leq i+j \leq r$, then $i \leq r$.

$$f(i,j) = (1-\phi)i - \phi j + \epsilon(n-(i+j)) \leq (1-\phi)i + \epsilon(n-i) \leq (1-\phi-\epsilon)i + \epsilon n \tag{7}$$

$$\leq (1-\phi-\epsilon)r + \epsilon n = (1-\phi)r + \epsilon(n-r) = f(r,0) \tag{8}$$

As $\phi \leq 0.5$, then $\phi \leq 1-\phi$, and as As $0 \leq i+j \leq r$, then $i \leq r$

$$\phi j - (1-\phi)i + \epsilon(n-(i+j)) \leq (1-\phi)j + \epsilon(n-j) \leq (1-\phi)r + \epsilon(n-r) \tag{9}$$

□

We next analyse the same range for $0.5 < \phi \leq 1$.

Claim 3. *For $0.5 < \phi \leq 1$ and $i, j > 0$ such that $0 \leq i+j \leq r$ and $\epsilon < 0.5$, then: (1) $\phi i - (1-\phi)j + \epsilon(n-(i+j)) \leq \phi r + \epsilon(n-r)$, and (2) $(1-\phi)i - \phi j + \epsilon(n-(i+j)) \leq \phi r + \epsilon(n-r)$.*

Proof. As $\phi > 0.5$, and $\epsilon \ll 0.5$ then $\phi - \epsilon > 0$. As $0 \leq i+j \leq r$, then $i \leq r$.

$$f(i,j) = \phi i - (1-\phi)j + \epsilon(n-(i+j)) \leq \phi i + \epsilon(n-i) \leq (\phi-\epsilon)i + \epsilon n \leq \phi r + \epsilon(n-r) = f(r,0) \tag{10}$$

As $\phi > 0.5$, then $(1-\phi) \leq \phi$, and as As $0 \leq i+j \leq r$, then $i \leq r$

$$(1-\phi)i - \phi j + \epsilon(n-(i+j)) \leq \phi i + \epsilon(n-i) \leq \phi r + \epsilon(n-r) \tag{11}$$

□

Putting the two claims together we get:

Claim 4. *For $0 \leq \phi \leq 1$ and $i, j > 0$ such that $0 \leq i+j \leq r$ and $\epsilon \ll 0.5$, then: (1) $\phi i - (1-\phi)j + \epsilon(n-(i+j)) \leq r + \epsilon(n-r)$, and (2) $(1-\phi)i - \phi j + \epsilon(n-(i+j)) \leq r + \epsilon(n-r)$.*

Proof. From Claim 2, for $0 \leq \phi \leq 0.5$ then both inequalities are bounded by $(1-\phi)r + \epsilon(n-r)$, and as $\phi \geq 0$ then $(1-\phi)r + \epsilon(n-r) \leq r + \epsilon(n-r)$.

From Claim 3, for $0.5 < \phi \leq 1$ then both inequalities are bounded by $\phi r + \epsilon(n-r)$, and as $\phi \leq 1$ then $\phi r + \epsilon(n-r) \leq r + \epsilon(n-r)$. □

Finally, we prove a bound on the rank of the element returned.

Lemma 10. *Given parameters (ϵ, δ) if $\epsilon < 0.5$, then the r-relaxed quantiles sketch returns an element whose rank is between $(\phi - \epsilon_r)n$ and $(\phi + \epsilon_r)n$ with probability at least $1-\delta$, where $\epsilon_r = \epsilon - \frac{r\epsilon}{n} + \frac{r}{n}$.*

Proof. Given parameters (ϵ, δ), and given that the adversary hides 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 is in the range given in Equation 2 w.p. at least $1-\delta$:

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

From Claim 4, this range is contained within the range:

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

Which can be rewritten as the range $\left(\phi \pm \left(\epsilon - \frac{r\epsilon}{n} + \frac{r}{n}\right)\right) n$. Meaning the rank of the element returned is between $(\phi - \epsilon_r)n$ and $(\phi + \epsilon_r)n$ with probability at least $1-\delta$, where $\epsilon_r = \epsilon - \frac{r\epsilon}{n} + \frac{r}{n}$. □