Fast Concurrent Data Sketches*

Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel
Idit Keidar, Lee Rhodes, Hadar Serviansky

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 some 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

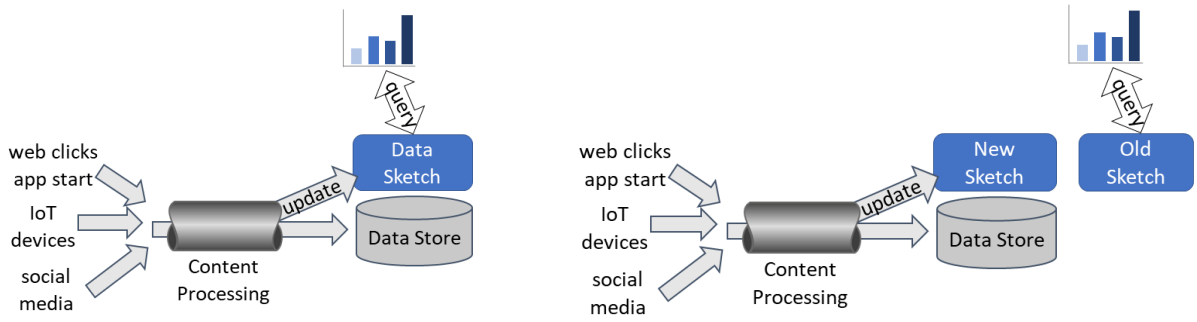
Motivation. Data sketching algorithms, or *sketches* for short [17], have become an indispensable tool for high-speed computations over massive datasets in recent years. Their applications include a variety of analytics and machine learning use cases, e.g., data aggregation [10, 14], graph mining [16], anomaly (e.g., intrusion) detection [34], real-time data analytics [21], and online classification [31].

Sketches are designed for *stream* settings in which each data item is only processed once. A common use case is data analytics, powered by analytics platforms like Druid [21]. A typical stream processing pipeline for data analytics is illustrated in Figure 1a. The stream consists of real-time events from various sources, such as web page clicks, apps being run on mobile devices, social media posts, and reports from IoT devices. The data is typically stored for archival purposes, and also summarised by data sketches to allow real-time queries. Another use case is network management [17], where statistics are gathered over a stream of network packets.

A sketch data structure is essentially a succinct (sublinear) summary of a stream that approximates a specific query, for instance, unique element count, quantile values, or frequent items. The approximation is typically very accurate – the error drops fast with the stream size [17].

Practical sketch implementations have recently emerged in toolkits [4] and data analytics platforms (e.g., PowerDrill [26], Druid [21], Hillview [7], and Presto [3]). 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 [8, 5]. As a consequence of this limitation, typical deployments create sketches in epochs, where queries are referred to the sketch created in the previous epoch while new stream elements are directed to a new sketch, as illustrated in Figure 1b. This practice leads to stale query results and thus loses the real-time quality of the system.

*An extended abstract describing this result appears in PPoPP 2020 with the same authors and title. The current version contains detailed correctness proofs that are absent in the extended abstract, in particular all of Section 6 and mathematical derivations in Section 7.



(a) Stream processing pipeline with data sketch.

(b) Using sketches in epochs.

Our approach. We present a generic approach to parallelising data sketches efficiently while bounding the error that such a parallel implementation might induce. Our goal is to enable simultaneous queries and updates to the same 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 [17], which enables computing a sketch over a stream by merging sketches over sub-streams. Previous works have exploited this property for distributed stream processing (e.g., [26, 19]), 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. Our solution architecture is illustrated in Figure 2. Multiple worker thread buffer updates in local sketches and periodically merge them into a global sketch; queries access the latter.

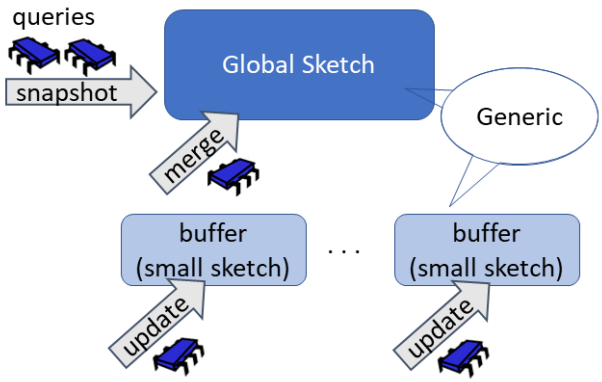


Figure 2: Concurrent sketches architecture.

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 a KMV Θ sketch [14], which estimates the number of unique elements in a stream; a popular sketch from the open-source Apache DataSketches library [4]. We have contributed our implementation back to the Apache DataSketches library [6]. Yet we

emphasize that our design is generic and applicable to additional sketches. We briefly discuss the applicability of our algorithm to additional popular sketches, such as Quantiles, CountMin, and HyperLogLog, where we discuss the generic algorithm (cf. Section 5).

Figure 3 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.

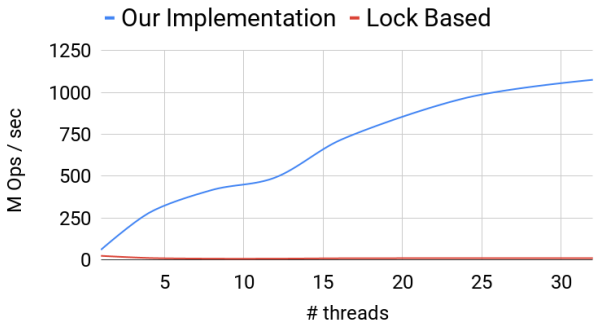


Figure 3: Scalability of DataSketches' Θ sketch protected by a lock vs. our concurrent implementation.

Error analysis. Concurrency might induce an error, and one of the main challenges we address is analysing this 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 [25, 11, 32] 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 [25] and priority queues [12, 29], 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. For instance, in a stream consisting of ten million events, missing a hundred (or even a thousand) of them will not make a big impact.

Analytics platforms often use multiple sketches in order to capture different dimensions of the data. For instance, they may count the number of unique users from each region in a different sketch. Typically, a handful of popular sketches account for most events, and others are updated less frequently. Whereas the relaxation does not significantly affect the estimation in the popular sketches, since the error allowed by the relaxation is additive, in less popular sub-streams, it may have a large impact. This motivates our adaptive solution, which forgoes relaxing small streams altogether.

We show that under parallel ingestion, our algorithm satisfies relaxed consistency with a relaxation of up to $2Nb$, where N is the number of worker threads and b is the buffer size of each worker. In our example use case, N is 12 and b ranges between 1 and 5.

The proof involves some technical challenges. First, 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* [24] with regards to a relaxed sequential specification of the de-randomised S. We note, however, that supporting strong linearisability did not incur additional costs nor did it impact the relaxation; we were able to prove that our original design was strongly linearisable.

We then analyse the error for two types of relaxed sketches under random coin flips, with an adversarial scheduler that may delay operations in a way that maximises the error. First, we consider the Θ sketch. For this sketch, its relative standard error has been analysed, and we show that our concurrent implementation's error is coarsely bounded by twice that of the corresponding sequential sketch. Second, we consider a family of *probably approximately correct (PAC)* sketches – these are sketches that estimate some quantity with an error of at most ϵ with a probability of at least $1 - \delta$. For an arbitrary PAC sketch estimating quantiles or counting unique elements, we show that the error induced by its relaxation approaches that of the original, non-relaxed sketch 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 proves strong linearisability of our generic algorithm. Section 7 analyses error bounds for example sketches. Section 8 empirically studies the Θ sketch's performance and error with different stream sizes. Finally, Section 9 concludes.

2 Model

We consider a non-sequentially consistent shared memory model that enforces program order on all variables and allows explicit definition of *atomic* variables as in Java [1] and C++ [15]. 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, where a step can access a shared memory variable, do local computations, and possibly return some value. 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 a special *invoke* step and ending in a *response* step that may return a value. 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. [24] have shown that in order to ensure correct behaviour of randomised algorithms under concurrency, one has to prove *strong linearisability*:

Definition 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, $A||A'$.

Algorithm 1 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
```

Example: Θ sketch. Our running example is a Θ sketch based on the *K Minimum Values (KMV)* algorithm [14] given in Algorithm 1. 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, 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 $1/\sqrt{k-2}$ [14].

A 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 $1-\delta$ [10].

4 Relaxed consistency for concurrent sketches

Previous work by Alistarh et al. [11] 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 [25] *out-of-order* relaxation, which generalises quasi-linearisabilty [9]. 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.

Definition 2 (r-relaxed history)**.** *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'.

A relaxed property for an object o is an extension of its sequential specification to include also relaxed histories and thus 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 3 (r-relaxation)**.** *The* r-relaxation *of SeqSketch is the set of histories that have r-relaxations in SeqSketch:*

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

Note that our formalism slightly differs from that of [25] 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.

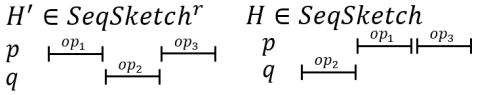


Figure 4: H is a 1-relaxation of H'.

An example is given in Figure 4, 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. That is, any sketch exposing this extended API can be used within our 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. 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. In general, 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, for instance, CountMin [18], HyperLogLog [23, 26, 21, 3], and Quantiles [27], atomic snapshots can be achieved in a straightforward manner via a double collect of the relevant state, e.g., array of counters. In specific sketches, this may be optimized in different ways.

In recent work [30], we have shown that for PAC objects, a linearisable snapshot is often not necessary for preserving the sketch's error bounds. We defined a relaxation of linearisability, called *Intermediate Value Linearisability (IVL)*. We proved that for any sequential PAC object – that is, one guaranteeing an error of at most ϵ with a probability of at least $1 - \delta$ for some parameters ϵ and δ – any concurrent implementation of this object that satisfies IVL guarantees the same ϵ, δ error bounds as the sequential object. In many cases, this allows replacing the linearisable snapshot with a single collect of the data structure, which is an array of counters in sketches like CountMin and HyperLogLog. In such cases, the implementation of the snapshot function is identical to the sequential sketch's query operation, and no synchronisation is required.

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 [33]. 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 and a stream element a, returns a Boolean indicating whether a should be added to the sketch, or may be filtered out as it does not affect the sketch's state.

Formally, the semantics of these APIs are defined using the notion of summary. (1) Consider a sketch S initialized in some state s_0. We say that s_0 (or the sketch at time 0) *summarises* the empty history, and similarly, the empty stream; we refer to the sketch as *empty*. (2) Let s' be the sketch's state after we sequentially ingest a stream $a_1, \ldots, a_n$, namely after a sequential execution with the history

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

We say that s' *summarises* history H, and, similarly, summarises the stream $a_1, \ldots, a_n$.

Given a sketch state s' that summarises a stream A, if shouldAdd($S.calcHint()$, a) returns false then for every streams B_1, B_2 and sketch state s' that summarises $A||B_1||a||B_2$, s' also summarises $A||B_1||B_2$. Note that a state summarizes two different streams if and only if that state is reached after ingesting each of them to an empty sketch.

These APIs do not need to support concurrency, and may be trivially implemented by always returning *true*. Thus, they do not impose additional constraints on the set of sketches usable with our generic algorithm.

Example: composable Θ sketch. Algorithm 2 presents the three additional APIs for the Θ sketch. The composable sketch is used concurrently by a single updater thread and multiple query threads. The *est* variable is atomic, and is shared among all threads; the remaining state variables are local to the updating thread.

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 2 Additional methods for composable Θ sketch.

```
1: procedure SNAPSHOT
2:     localCopy ← emptysketch
3:     localCopy.est ← est
4:     return localCopy
5: procedure CALCHINT
6:     return Θ
7: procedure SHOULDADD(H, arg)
8:     return h(arg) < H
```

5.2 Generic algorithm

We now present a generic concurrent sketch algorithm that can be instantiated with any composobale sketch adhering to the API defined in the previous section. To simplify the presentation, we first discuss an unoptimised version of our generic concurrent algorithm, (left column in of Algorithm 3), called *ParSketch*, and later an optimised version of the same algorithm (right column of Algorithm 3).

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.

Algorithm 3 Generic concurrent algorithm.

Basic 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, init empty
106:
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)
114:             localS_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.update(a)
123:     if counter_i = b then
124:         prop_i ← 0
125:         wait until prop_i ≠ 0
126:
127:         hint_i ← prop_i
128:         counter_i ← 0
129:
```

Optimised algorithm

```
201: variables
202:     composable sketch globalS, init empty
203:     constant b                          ▷ relaxation is 2Nb
204:     for each update thread t_i , 0 ≤ i ≤ N
205:         sketch localS_i[2], init empty
206:         int cur_i, init 0
207:         int counter_i, init 0
208:         int hint_i, init 1
209:         int atomic prop_i, init 1
210: procedure PROPAGATOR
211:     while true do
212:         for all thread t_i s.t. prop_i = 0 do
213:             globalS.merge(localS_i[1-cur_i])
214:             localS_i[1 − cur_i] ←empty sketch
215:             prop_i ← globalS.calcHint()
216: procedure QUERY(arg)
217:     localCopy ← globalS.snapshot(localCopy)
218:     return localCopy.query(arg)
219: procedure UPDATE_i(a)
220:     if ¬shouldAdd(hint_i, a) then return
221:     counter_i ← counter_i + 1
222:     localS_i[cur_i].update(a)
223:     if counter_i = b then
224:
225:         wait until prop_i ≠ 0
226:         cur_i ← 1 − cur_i
227:         hint_i ← prop_i
228:         counter_i ← 0
229:         prop_i ← 0
```

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 Section 6.2 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 with regards to* SeqSketch^r*, where* $r = 2Nb$.

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. In the right column of Algorithm 3, we present 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 226), and immediately sets $prop_i$ to 0 (line 229) in order to allow the propagator to take the information from the other one. It then starts digesting updates in a fresh sketch.

Of course, the optimisation is only useful as long as the propagator thread is fast enough to empty the inactive buffers before the active ones fill up. The number of threads where this will saturate is highly sketch-dependant. In the example of the Θ sketch, thanks to pre-filtering, the working threads filter out many updates without filling their buffers, so merges are required infrequently, and we can scale to a large number of threads with a single propagator regardless of the buffer size. In sketches without pre-filtering, the scalability typically depends on the buffer size.

In Section 6.3 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* [28]. 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 with regards to* SeqSketch^r*, where* $r = 2Nb$.

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 3 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 correctness of the adaptation is straightforward, since the buffer size is only used locally and only impacts the relaxation. The error analysis of the next section can be used to determine the adaptation point.

6 Proofs

In Section 6.1 we introduce some formalisms. In Section 6.2 we prove that the unoptimised algorithm is strongly linearisable with respect to the relaxed specification $SeqSketch^r$ with $r = Nb$. Finally, in Section 6.3 we show that the the optimised algorithm is strongly linearisable with respect to the relaxed specification $SeqSketch^r$ with $r = 2Nb$.

6.1 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 auxiliary 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's state *summarises* a stream or a sequential history if it 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 4 (Thread summary)**.** *Consider a time t in an execution σ of Algorithm 3. 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.*

Note that if the first condition ($prop_i \neq 0$ or $sig\_ctr[i] > merge\_ctr[i]$) is not satisfied, this means that the propagator thread might be in the process of clearing $localS_i$ in line 114.

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 5 (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 6 (Unprop updates)**.** *Given a finite execution σ, we denote by $suff_i(\sigma)$ 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 $up\_suff_i(\sigma)$ of update thread i is the subsequence of $\mathcal{H}(suff_i(\sigma))$ consisting of update(a) executions in $suff_i(\sigma)$ 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 7. *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 8. *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.*

6.2 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 update$_i(a)$ where shouldAdd($prop_i$, a) returns false at time t, its visibility point is t.
- Otherwise, for an 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 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* defined 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 only 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 to 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 4, 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, t_i summarises the history up_suff$_i(\sigma')$. Because before the step $prop_i \neq 0$, $localS_i$ summarises the same history. 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 4, 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 6, 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 4, 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, by Definition 4, $localS_i$ 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 that step $prop_i \neq 0$. By Definition 4, 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 by 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 $\text{up\_suff}_i(\sigma)$. We denote the number of updates in history H as $|H|$.

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

Proof. As $counter_i$ is incremented before an update which is included in $\text{up\_suff}_i(\sigma)$, it follows that $|\text{up\_suff}_i(\sigma)| \leq counter_i$. 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, $count_i \leq b$. Therefore, it follows that $|\text{up\_suff}_i(\sigma)| \leq b$. □

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

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

Proof. From Lemma 5, $|\text{up\_suff}_i(\sigma)| \leq b$. 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 $|f(\sigma)| \geq |l(\sigma)| - Nb$. □

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

Lemma 7. *Given a finite execution σ of ParSketch, $f(\sigma)$ is an Nb-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 $|Ops| \leq Nb$. 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, $|f(\sigma^{pre})| \geq |l(\sigma^{pre})| - Nb$. 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 Nb 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, $|Ops| = |l(\sigma^{pre})| - |f(\sigma^{pre})| \leq Nb$. Therefore, by Definition 3, $f(\sigma)$ is an Nb-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 Nb-relaxation of $l(\sigma)$. We have proven Lemma 1.

6.3 Optimised algorithm proof

We denote the optimised version of Algorithm 3 as *OptParSketch*. We prove the correctness of *OptParSketch* by showing that it can simulate *ParSketch*. This proof technique is known as a *simulation relation*, which, as explained in [28], Chapter 2.5, is a correspondence relating the states of *OptParSketch* and *ParSketch* when the algorithms run on the same input stream. Establishing a simulation relation proves that $OptParSketch$ is strongly linearisable with regards to $SeqSketch^{2Nb}$ [22, 13].

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 the basic algorithm. To simulate N worker threads, we need $2N$ threads, and they are mapped the same way.

The idea behind the simulation is that there might be a delay between the time when the *hint* is returned to the worker thread and the time when this hint is used for pre-processing, so we can simulate each thread by two threads. For example in Figure 5, 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 $A = A_1||A_2||\dots||A_n$ and we assume n is even. Each A_i is evaluated against the *hint* written above it. The thread t_i^0 simulates processing $A_1||A_3||\dots||A_{n-1}$, and thread t_i^1 simulates processing $A_2||A_4||\dots||A_n$.

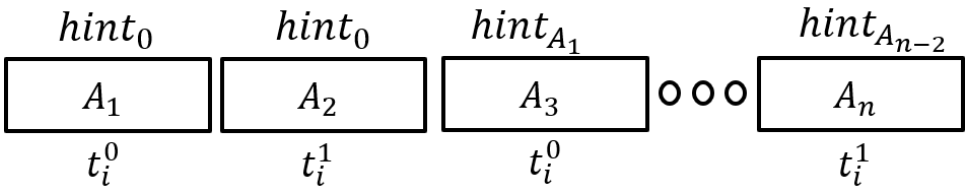


Figure 5: Simulation of processing $A = A_1||A_2||\ldots||A_n$.

The simulation uses auxiliary variables oldHint$_i^0$, and oldHint$_i^1$, both initialised to 1. These variables are updated with the flipping of cur_i (line 226), such that:

- oldHint$_i^0$ is updated to be the current (pre-flip) value of $hint_i$
- oldHint$_i^1$ is updated to be the current (pre-flip) value of $oldHint_i^0$

In addition, the simulation uses an auxiliary variable $auxCount_i$ initialised to 0. This variable is set to b before the first execution of line 226, and is never changed after that.

Finally, the simulation uses two auxiliary 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*.

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 $ParSketch$.
- 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 227, otherwise oldHint$_i^0$ 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 227-229, 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 6 shows a mapping when cur_i equals 0, before executing line 227. Table 1 shows the steps taken by t_i^0 and t_i^1 when $cur_i = 0$ before line 223.

OptParSketch line	*ParSketch* line	Executing thread
223	123	t_i^0
225	125	t_i^1
226	-	-
227	127	t_i^1
228	128	t_i^1
229	124	t_i^0

Table 1: 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 223, meaning the "round" of b updates was ingested by t_i^0. On line 226 neither thread takes a step.

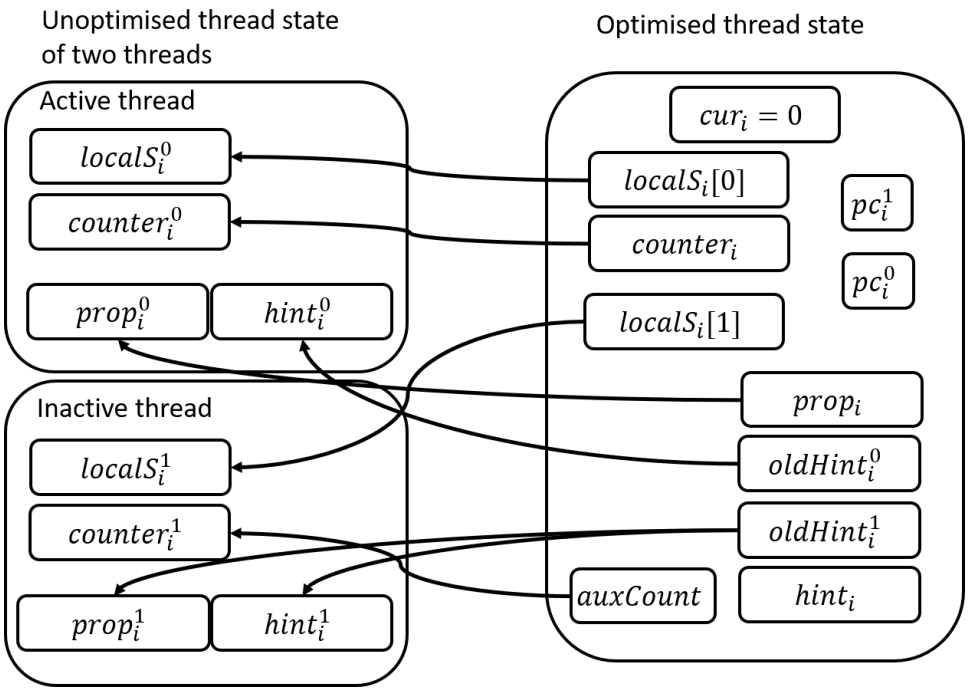


Figure 6: Reference mapping of g when cur_i equals 0 before executing line 227.

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 *update* is invoked in $t_i^{cur_i}$ in *ParSketch*. If the counter gets up to b (meaning we get to line 225), then $t_i^{1-cur_i}$ executes line 125. When *OptParSketch* flips cur_i (line 226), then neither of the threads t_i^0 or t_i^1 take a step. Afterwards, lines 227 and 228 execute the corresponding lines (127 and 128) on thread $t_i^{cur_i}$, and line 229 executes 124 on thread $t_i^{1-cur_i}$.

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

Proof. The proof is by induction on the steps in an execution, for some thread i. In the initial state, the mapping trivially holds. In a given step, we refer to $t_i^{cur_i}$ as the *active* thread and $t_i^{1-cur_i}$ as the *inactive thread*. Query threads trivially map to themselves and do not alter the state. We next consider update and propagator threads. First, consider the steps of OptParSketch that execute the corresponding step on the active thread. These are lines 219–223 and 227–228, which directly correspond to lines 119-123 and 127-128 of ParSketch in the active thread ($t_i^{cur_i}$), and, except in lines 127 and 129, the effected 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 229. As after this step the inactive thread's prop and prop_i are both 0, so 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 226 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^0 and oldHint_i^1 are updated as defined, and as t_i is right before executing line 227, oldHint_i^1 is equal to the inactive thread's ($t_i^{1-cur_i}$) 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^0, after this step it is equal to oldHint_i^1. 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 227, 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 228 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 210-215) 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". This establishes strong linearisability [13], intuitively, because 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 with regards to* SeqSketch^r*, where* $r = 2Nb$.

7 Deriving error bounds

We now show how to translate the r-relaxation to a bound on the error of typical sketches. We consider two types of error anlayses of existing sketches. In Section 7.1, we consider the relative standard error of the Θ sketch, which was used in the original analysis of the sketch. In Section 7.2.1 we consider PAC sketches, and show generic error bounds for all r-relaxed implementations of PAC sketches estimating the number of unique elements and quantiles.

7.1 Θ error bounds

We bound the error introduced by an r-relaxation of the Θ sketch over a stream with n unique elements and a parameter (sketch size) of k. 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 2. Our analysis relies on known results from order statistics [20]. It focuses on long streams, and assumes $n > k + r$.

	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 2: Expectation and RSE of Θ sketch with numerical values for $r = 8, k = 2^{10}, n = 2^{15}$.

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 2). The sequential (non-relaxed) sketch returns $e = est(M_{(k)})$. It has been shown that the sketch is unbiased [14], i.e., $E[e] = n$ the number of unique elements. Moreover, previous work [2] has analysed the *relative standard error (RSE)* of the sketch, which is the standard error divided by the mean, and has shown it to be $RSE[e] \leq \frac{1}{\sqrt{k-2}}$.

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 so it cannot distinguish the algorithm state (set of retained elements) from a random one. Since the state is random in all runs, it chooses the same j in all runs. We show that the largest error is always obtained either for $j = 0$ or for $j = r$.

Claim 1. *Consider j values X_i, $1 \leq i \leq j$, 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 $\frac{k-1}{M_{(i)}} \leq n$, as $M_{(j)} > M_{(i)}$ then $\frac{k-1}{M_{(j)}} < \frac{k-1}{M_{(i)}} \leq n$, 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.

□

Consider an adversary $\mathcal{A}$ whose estimate is a random variable $e_{\mathcal{A}}$, characterized by the probability density function $f_{e_{\mathcal{A}}}$. The expectation of $e_{\mathcal{A}}$ is not necessarily n, and so the relative standard error needs to be computed as the error from the desired estimate, n, rather than from the expectation. This can be done using the following formula:

$$(RSE[e_{\mathcal{A}}])^2 = \frac{1}{n^2} \int_{-\infty}^{\infty} (e - n)^2 \cdot f_{e_{\mathcal{A}}}(e)\, de$$

We prove the following bound:

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

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

Proof.

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

□

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:

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

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, 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:

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

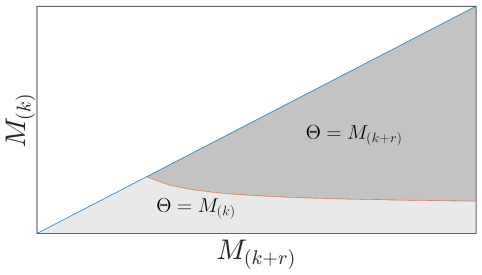


Figure 7: Areas of $M_{(k)}$ and $M_{(k+r)}$. In the dark gray $\mathcal{A}_s$ induces $\Theta = M_{(k+r)}$, and in the light gray, $\Theta = M_{(k)}$. The white area is not feasible.

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

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

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

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

In Figure 7 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 7 using their joint probability function from order statistics. Equations 1 and 2 give the formulas for the expected estimate and its RSE bound, respectively. We do not have closed-form bounds for these equations. Example numerical results, computed based on Equation 2, are shown in Table 2.

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 8 depicts the distribution of $e_{\mathcal{A}_w}$, and the distribution of e is shown in blue.

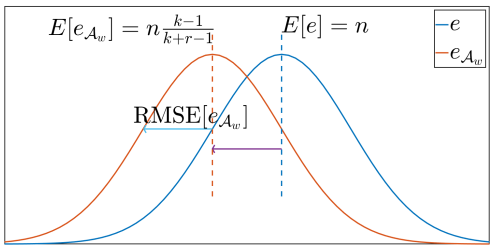


Figure 8: Distribution of estimators e and $e_{\mathcal{A}_w}$. The RSE of $e_{\mathcal{A}_w}$ with regards to n is bounded by the relative bias plus the RMSE of $e_{\mathcal{A}_w}$.

Recall that $\mathcal{A}_w$ always hides r elements smaller than Θ, thus forcing $M^r_{(k)} = M_{(k+r)}$. Here too our analysis is on the order statistics for 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 the following equation:

$$\sqrt{\frac{\sigma^2[e_{\mathcal{A}_w}]}{E[e_{\mathcal{A}_w}]}} < \frac{1}{k-2} \tag{3}$$

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 2:

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

Using Equation 3:

$$\mathrm{RSE}[e_{\mathcal{A}_w}] < \sqrt{\frac{1}{k-2}} + \frac{r}{k-2} \tag{4}$$

We have shown that the RSE is bounded by $\sqrt{\frac{1}{k-2}} + \frac{r}{k-2}$ for $\mathcal{A}_w$. 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.

7.2 Error bounds for PAC sketches

We now provide a generic analysis, considering a PAC sketch as a black box. Section 7.2.1 studies quantiles sketches, and in Section 7.2.2, we study PAC sketches estimating the number of unique elements in a stream, e.g., HyperLogLog. In both cases, we show that if the sequential sketch's error bound is ϵ, then the error of an r-relaxed sketch over a stream of size n is bounded by $\epsilon + \frac{r\epsilon}{n} + \frac{r}{n}$. This expression tends to ϵ as the stream sizes grows to infinity, but may be substantially larger for small streams. A system designer can use this formula to determine the adaptation point so that the error is never above a desired threshold.

7.2.1 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., [10]) are sensitive to the processing order. In this case, advance 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 the original stream is in the range:

$$[(\phi - \epsilon)(n - (i + j)) + i, (\phi + \epsilon)(n - (i + j)) + i]\,. \tag{5}$$

This can be rewritten as:

$$\begin{gathered}[\phi n - (\phi j - (1 - \phi)i + \epsilon(n - (i + j))), \\ \phi n + ((1 - \phi)i - \phi j + \epsilon(n - (i + j)))]\end{gathered} \tag{6}$$

Note that this interval is symmetric around $\phi(n - (i + j)) + i$. The adversary attempts to maximise the distance of the edges of this interval from the true rank, (i.e., maximise ϵ_r). The distance between the central points is:

$$|\phi n + (1 - \phi)i - \phi j - \phi n| = |(1 - \phi)i - (\phi)j|.$$

Given that $0 \leq i + j \leq r$, we show that this expression is maximised for $i + j = r$.

Claim 2. *Given $0 \leq i, j$ such that $0 \leq i + j \leq r$, the expression $|(1 - \phi)i - (\phi)j|$ is maximised for $(i, j) = (x, y)$ such that $x + y = r$.*

Proof. Assume by contradiction that the expression given in the claim is maximised for (x, y) such that $x + y = r' < r$. Denote $r' = r - k$. We consider two cases for the expression $(1 - \phi)i - (\phi)j$.

If $(1 - \phi)x - (\phi)y \geq 0$, then $(1 - \phi)(x + k) - (\phi)y \geq (1 - \phi)x - (\phi)y > 0$. In this case denote $x' = x + k$ and $y' = y$.

If $(1 - \phi)x - (\phi)y < 0$, then $(1 - \phi)x - (\phi)(y + k) \leq (1 - \phi)x - (\phi)y < 0$. In this case denote $x' = x$ and $y' = y + k$.

In both cases we found (x', y') such that $x' + y' = r$ and the expression $|(1 - \phi)i - (\phi)j|$ is maximised for $(i, j) = (x', y')$. □

By substituting $j = r - i$ into the error formula, we get:

$$|(1 - \phi)i - (\phi)(r - i)| = |i - \phi r|.$$

As $0 \leq \phi \leq 1$, the following claim follows immediately:

Claim 3. *For $\phi \leq 0.5$ the adversary maximises the distance by choosing $i = r$ (and therefore $j = 0$) and for $\phi > 0.5$ the adversary maximises the error by choosing $i = 0$ (and therefore $j = r$).*

We begin by analysing the range given in Equation 6 for $0 \leq \phi \leq 0.5$.

Claim 4. *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 5. *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 6. *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 4, 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 5, 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 6 w.p. at least $1 - \delta$:

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

From Claim 6, 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 $\epsilon_r = \epsilon - \frac{r\epsilon}{n} + \frac{r}{n}$. □

We have shown 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.

7.2.2 Count unique elements error bounds

Finally, we consider the error of any implementation of a count unique elements sketch, provided that the sketch is PAC. In this case, for a stream with n unique elements, the query returns an estimate e which is in between $(1 - \epsilon)n$ and $(1 + \epsilon)n$ with probability at least $1 - \delta$ for some parameters ϵ and δ. We show that the r-relaxation of such a sketch returns an estimate is in the range $(1 \pm \epsilon_r)n$ with probability at least $1 - \delta$ for $\epsilon_r = \epsilon + \frac{r\epsilon}{n} + \frac{r}{n}$.

As in a Quantiles sketch, advance knowledge of the coin flip can increase the error already in the sequential sketch. Therefore, here too, we focus on a weak adversary. As above, the adversary hides either no elements or r elements. If the adversary hides r elements, the estimate returned is in the range $(1 \pm \epsilon)(n - r)$.

The adversary thus chooses whether to hide r elements or not based on which estimate maximises the error $|n - e|$. In either case, with probability at least $1 - \delta$ the estimate is between $(1 - \epsilon)(n - r)$ and $(1+\epsilon)n$. This range is contained in the range $n\left(1 \pm \left(\epsilon + \frac{r\epsilon}{n} + \frac{r}{n}\right)\right)$. We can define $\epsilon_r \triangleq \epsilon + \frac{r\epsilon}{n} + \frac{r}{n}$. Note that, as in the case of the Quantiles sketch, here too, the impact of the relaxation diminishes as n grows.

8 Θ sketch evaluation

This section presents an evaluation of an implementation of our algorithm for the Θ sketch. Section 8.1 presents the methodology for the analysis. Section 8.2 shows the results under different workloads and scenarios. Finally, Section 8.3 discusses the tradeoff between accuracy and throughput.

8.1 Setup and methodology

Our implementation [6] extends the code in Apache DataSketches [4], 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 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.

We implement a limit for eager propagation as a function of the configurable error parameter ϵ; the function we use is $2/\epsilon^2$. The local sketches define b as a function of k, ϵ, and N (the number of

writer threads) such that the error induced by the relaxation when in the lazy propagation mode does not exceed e using Equation 4. Thus the total error is bounded by $\max\{\epsilon + \frac{1}{\sqrt{k}}, \frac{2}{\sqrt{k}}\}$.

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 stated otherwise, we use k=4096, which is commonly used [4] for the Θ sketch. The sequential sketch's RSE with this buffer size is 0.031 with a probability of at least 0.95. In the concurrent sketch, we chose to limit the error to $\epsilon = 0.04$ with the same probability. Given a particular number of threads N, b is derived according to Equation 4 with $r = 2Nb$. Recall that the analysis in Section 7.1 (including this equation) is conditioned on the assumption that $n > k+r$. Therefore, if we would set the eager adaptation threshold to $k + 2Nb$, we would get the same error bound for any sketch size. However, this is a conservative choice. We experiment with a threshold of 1250, and show that empirically, the error is reasonable with this choice. In general, this is a configurable parameter, which can be used by system designers to navigate the tradeoff between accuracy and performance.

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 simulate 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. Small stream sizes tend to suffer more from measurement noise, so the number of trials is very high (in the millions). As the stream size gets larger, the number of trials gradually decreases down to 16 in the largest stream. .

Note that accuracy is measured relative to the number of unique elements ingested to the sketch before a query in some linearisation; because we cannot empirically deduce the linearisation point of a query that is run in parallel with updates, the metric is only well-defined when the query is not concurrent to any update. Therefore, we measure accuracy only in a single-thread environment, where we periodically interleave queries with updates of the same thread. The accuracy with more threads can be extrapolated from these measurements based on the theoretical analysis.

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".

8.2 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 9a. There are two interesting phenomena worth noting. First, it is interesting to see empirical evaluation reflecting the theoretical analysis presented in Section 7.1, 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 9b 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 10a presents throughput measurements for a write-only workload. The results are shown in log log scale. Figure 10b zooms-in on the throughput of large streams. As explained in Section 8.1, we compare the concurrent implementation to a lock-based approach. The number of threads in both implementations refers to the number of worker threads; there can be arbitrarily many reader threads.

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. At the peak measured performance the single threaded concurrent Θ sketch outperforms the single threaded 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 11 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 3 (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.

8.3 Accuracy-throughput tradeoff

The speedup achieved by eager propagation in small streams is presented in Figure 12. This is an additional advantage of eager propagation in small streams, beyond the accuracy benefit reported in Figure 9. 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 3 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 3: Performance vs accuracy as a function of k.

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

[2] Theta sketch equations. `https://github.com/apache/datasketches-website/blob/master/docs/pdf/ThetaSketchEquations.pdf`, 2015.

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

[4] Apache DataSketches. `https://datasketches.apache.org/`, 2019.

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

[6] DataSketches: Concurrent Theta Sketch Implementation. `https://datasketches.apache.org/docs/Theta/ConcurrentThetaSketch.html`, 2019.

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

[8] SketchesArgumentException: Key not found and no empty slot in table. `https://groups.google.com/d/msg/sketches-user/S1PEAneLmhk/dI8RbN6iBAAJ`., 2019.

[9] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proceedings of the 14th International Conference on Principles of Distributed Systems*, OPODIS'10, pages 395–410, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, pages 23–34, New York, NY, USA, 2012. ACM.

[11] Dan Alistarh, Trevor Brown, Justin Kopinsky, Jerry Z Li, and Giorgi Nadiradze. Distributionally linearizable data structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. ACM, 2018.

[12] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The spraylist: A scalable relaxed priority queue. *SIGPLAN Not.*, 50(8):11–20, January 2015.

[13] Hagit Attiya and Constantin Enea. Putting strong linearizability in context: Preserving hyperproperties in programs that use concurrent objects. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[14] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In Jos'e D. P. Rolim and Salil Vadhan, editors, *Randomization and Approximation Techniques in Computer Science*, pages 1–10, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[15] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, June 2008.

[16] Edith Cohen. 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, pages 88–99, New York, NY, USA, 2014. ACM.

[17] Graham Cormode. Data sketching. *Queue*, 15(2):60:49–60:67, April 2017.

[18] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

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

[20] Herbert Aron David and Haikady Navada Nagaraja. Order statistics. *Encyclopedia of Statistical Sciences*, 9, 2004.

[21] Druid. How We Scaled HyperLogLog: Three Real-World Optimizations. `http://druid.io/blog/2014/02/18/hyperloglog-optimizations-for-real-world-systems.html`.

[22] Ivana Filipovi, Peter OHearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51–52):4379–4398, December 2010.

[23] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and et al. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *International Conference on Analysis of Algorithms*, 2007.

[24] Wojciech Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 373–382. ACM, 2011.

[25] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *ACM SIGPLAN Notices*, volume 48, pages 317–328. ACM, 2013.

[26] Stefan Heule, Marc Nunkesser, and Alex Hall. Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm. In *Proceedings of the EDBT 2013 Conference*, Genoa, Italy, 2013.

[27] Z. Karnin, K. Lang, and E. Liberty. Optimal quantile approximation in streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 71–78, Oct 2016.

[28] Nancy A Lynch. *Distributed algorithms*. Elsevier, 1996.

[29] Hamza Rihani, Peter Sanders, and Roman Dementiev. Multiqueues: Simpler, faster, and better relaxed concurrent priority queues. *arXiv preprint arXiv:1411.1209*, 2014.

[30] Arik Rinberg and Idit Keidar. Intermediate Value Linearizability: A Quantitative Correctness Criterion. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

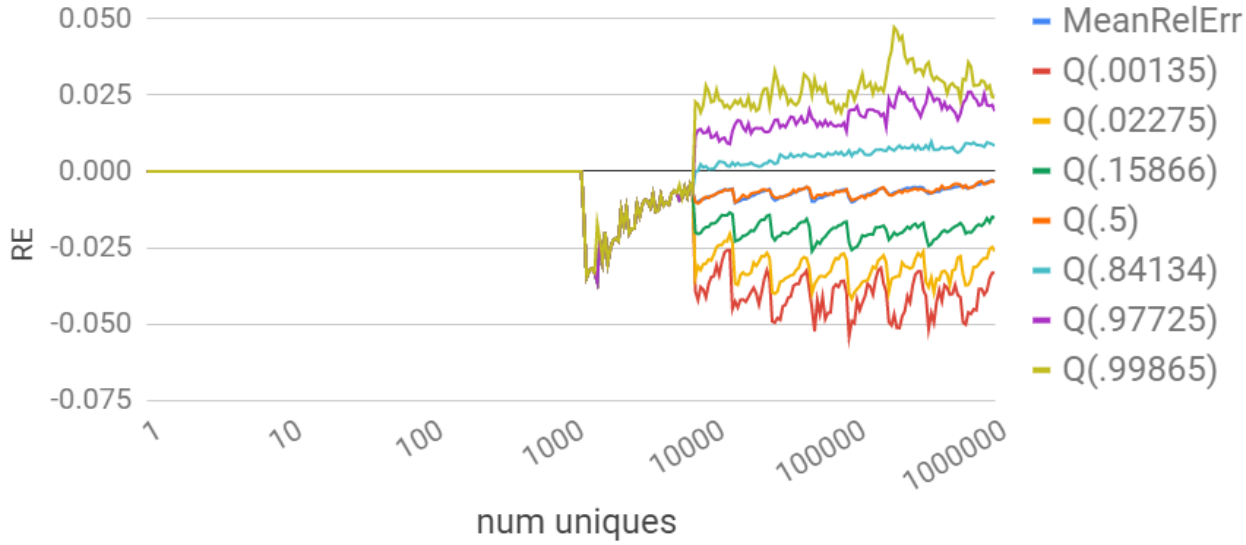
[31] Kai Sheng Tai, Vatsal Sharan, Peter Bailis, and Gregory Valiant. Sketching linear classifiers over data streams. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 757–772, New York, NY, USA, 2018. ACM.

[32] Edward Talmage and Jennifer L Welch. Improving average performance by relaxing distributed data structures. In *International Symposium on Distributed Computing*, pages 421–438. Springer, 2014.

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

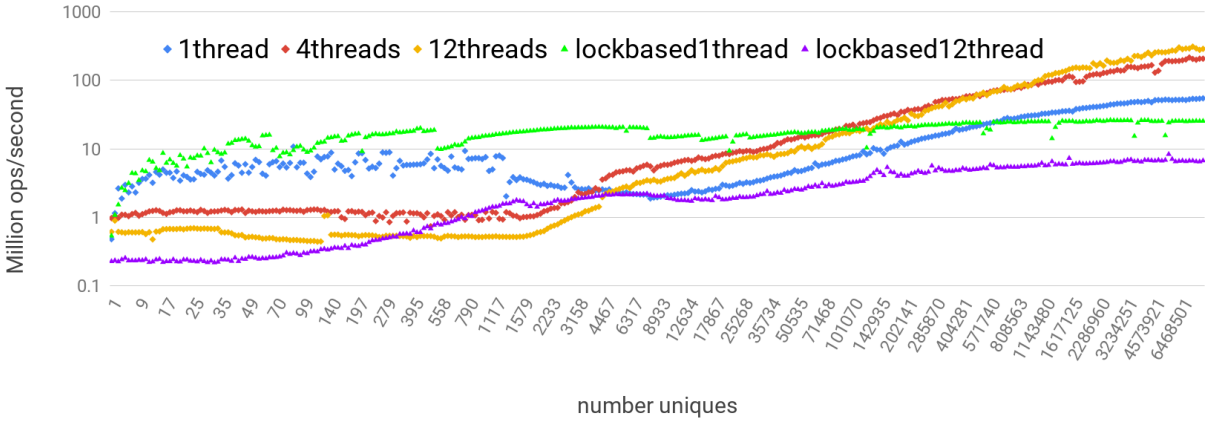
[34] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 561–575, New York, NY, USA, 2018. ACM.

(a) No eager propagation ($\epsilon = 1.0$)

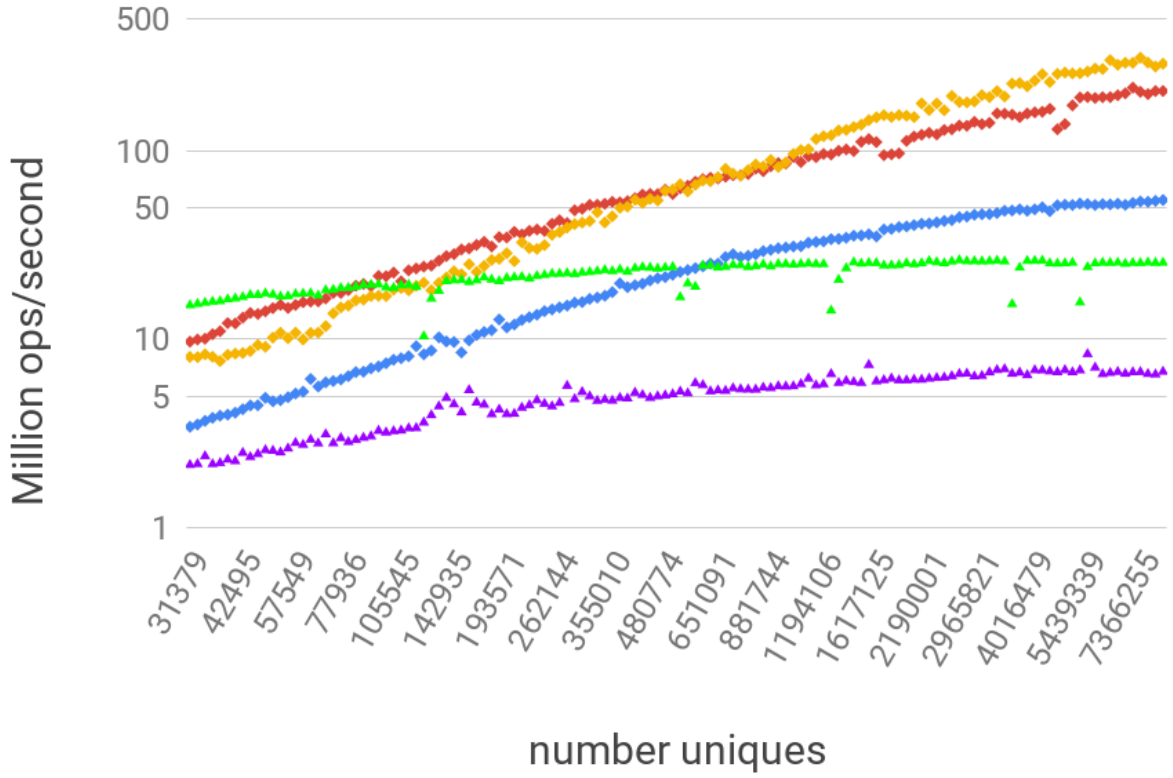


(b) With eager propagation, error bound defined by $\epsilon = 0.04$

Figure 9: Concurrent Θ measured quantiles vs RE, $k = 4096$.



(a) Throughput, loglog scale



(b) Zooming-in on large sketches

Figure 10: Write-only workload, $k = 4096$, $\epsilon = 0.04$.

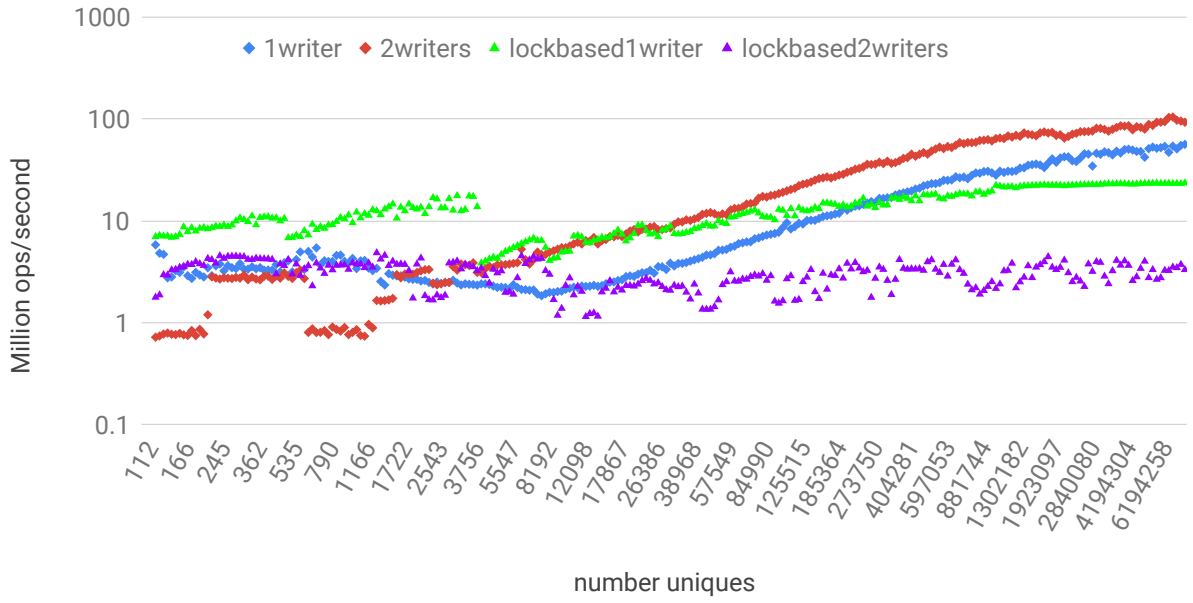


Figure 11: Mixed workloads: writers with background reads, $k = 4096$, $\epsilon = 0.04$.

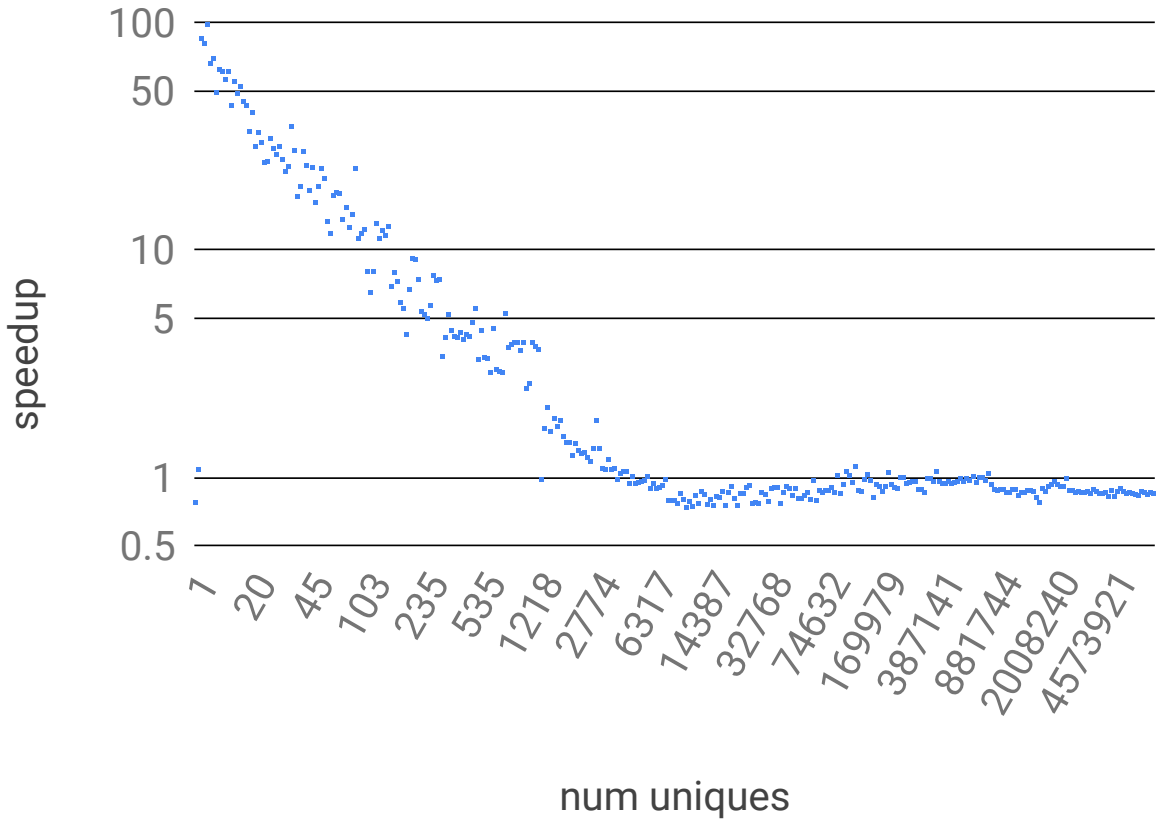


Figure 12: Throughput speedup of eager ($\epsilon = 0.04$) vs no-eager ($\epsilon = 1.0$) propagation, $k = 4096$.

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 3 – parseFigure1.py
- Figure 9 – 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-characterizatio`

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.