Concurrent Sketching: Theory and Practice

Arik Rinberg

Concurrent Sketching: Theory and Practice

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Arik Rinberg

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

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

The generous financial help of the Technion and the Gutwirth and Jacobs Fellowship is gratefully acknowledged.

To my loving wife, Noa, and wonderful children Rephael and Nesya.

List of Publications

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

1. Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky, "Fast concurrent data sketches", *ACM Transactions on Parallel Computing (TOPC) 9.2 (2022), pp. 1–35.*

   – An earlier version appeared in *Principles and Practice of Parallel Programming (PPoPP) 2020.*

2. Arik Rinberg and Idit Keidar, "Intermediate Value Linearizability: A Quantitative Correctness Criterion", To appear in *Journal of the ACM, accepted 19th January 2023.*

   – An earlier version appeared in *34th International Symposium on Distributed Computing 2020 (DISC'20).* **Best Student Paper.**

3. Dor Harris†, Arik Rinberg†, and Ori Rottenstreich, "Compressing Distributed Network Sketches with Traffic-Aware Summaries", *IEEE Transactions on Network and Service Management (2022)*, † - equal contributors.

   – An earlier version appeared in *International Federation for Information Processing (IFIP) Networking 2021.*

4. Shaked Elias-Zada, Arik Rinberg, and Idit Keidar, "Quancurrent: A Concurrent Quantiles Sketch"

   – Submitted for publication.

5. Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein, "SwiSh: Distributed Shared State Abstractions for Programmable Switches", *19th USENIX Symposium on Networked Systems Design and Implementation 2022 (NSDI'22), pp. 171–191.*

The author and research collaborators have also published papers that have not contributed to this thesis during the course of the author's doctoral research period, the most up-to-date versions of which being:

6. Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi, “ACE: Abstract Consensus Encapsulation for Liveness Boosting of State Machine Replication”, *24th International Conference on Principles of Distributed Systems 2021 (OPODIS’21).*

7. Arik Rinberg, Tomer Solomon, Guy Khazma, Gal Lushi, Roee Shlomo, and Paula Ta-Shma, “Array CRDTs using delta-mutations”, *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data 2021 (PaPoC’21), pp. 1–3.*

8. Arik Rinberg, Tomer Solomon, Roee Shlomo, Guy Khazma, Gal Lushi, Idit Keidar, and Paula Ta-Shma, “DSON: JSON CRDT using delta-mutations for document stores”, *Proceedings of the VLDB Endowment 15.5 2022 (VLDB’22), pp. 1053–1065.*

Acknowledgements

I would like to thank the amazing people who made my doctoral research period both fruitful and enjoyable.

- I would like to thank my advisor, Idit Keidar, for her amazing guidance and for teaching me to be a researcher. Thank you for being an inspiring role model, and for all your patience. It was my greatest pleasure to learn from you.
- I wish to thank Sasha, Eddy, Eschar, Lee, Hadar, Dor, Ori, Dahlia, Tomer, Guy, Gal, Roee, Paula, Lior, Dan, Jacob, Daehyeok, Shir, Alon, Igot, and Mark, for productive and enjoyable collaborations during my research.
- I would like to thank my research group's team members for the fun times during team meetings, during lunches, and during coffee breaks. Specifically, I would like to thank Gal, Ramy, and Shaked, for our professional discussions. I also thank Ramy and Shaked for being great office mates.
- I wish to thank Orly and Adi for their administrative support. Thank you for your dedication and speedy answers.
- I would like to thank my parents, Richard and Emma, for believing in me and for doing a great job raising me. Thank you for the encouraging words you had to offer whenever I needed them.
- I wish to thank my parents-in-law, Deborah and Elan, for their kind support and assistance.
- I would like to thank all my family, Aaron, Carmella, Maya, Lee, Lior, Rachel, Dean, Aria, Joseph, and Avishai, for making my breaks from the Technion enjoyable.
- I wish to thank my lovely children, Rephael ☺ and Nesya ☺, for providing me with pure joy and happiness.
- Finally, and most importantly, I wish to thank my darling wife Noa. Your support and sacrifice throughout my doctorate is what made it possible. I am constantly amazed by you, and am grateful to have you by my side.

Contents

List of Figures

Abstract

Processing high-speed, high-volume data often requires performing analytics on a representation of the data, rather than the data itself. To this end, data sketching algorithms have become indispensable tools for high-speed computations over massive datasets. They maintain the streams' state and answer queries on it (e.g., how many unique elements are in the stream) using limited memory, at the cost of giving approximations rather than exact answers. In this thesis, we study three aspects of these systems: (1) efficient concurrent implementation, (2) correctness semantics, and (3) efficient network implementation.

Existing libraries provide highly optimized sketches, but do not allow parallelism for creating sketches using multiple threads, or querying them while they are being built. Utilizing relaxed semantics and the notion of strong linearizability, we present a generic approach to parallelizing data sketches efficiently and allowing them to be queried in real-time. We prove its correctness and analyze its error in some specific sketches. We instantiate our algorithm with the KMV sketch, achieving terrific scalability with small error, which we then contribute to open source. However, when plugging in other sketches we identify two performance bottlenecks.

The first stems from the chosen correctness criterion. The de facto correctness criterion for concurrent objects is linearizability. Intuitively, under linearizability, a read overlapping an update must return the object's value either before or after the update. We observe, however, that in some cases, any *intermediate* value between the invocation and response would also be acceptable. To capture this degree of freedom, we propose *Intermediate Value Linearizability (IVL)*, a new correctness criterion that allows returning these intermediate values. Roughly speaking, IVL allows reads to return any value that is bounded between two return values that are legal under linearizability. We show that using IVL instead of linearizability allows for circumventing the first of our bottlenecks, and that IVL implementations of sketches inherit the error of their sequential counterparts.

The second bottleneck is due to the sequential propagation from thread-local sketches into the shared sketch. Such propagation is light in some sketches, e.g., KMV, but may be costly in others, e.g., Quantiles. The Quantiles sketch maintains a series of buffers, with propagations taking a variable amount of time depending on the size of the stream. To this end, we propose Quancurrent, a concurrent Quantiles sketch. We show how we design Quancurrent to provide a more scalable solution.

Lastly, we study sketches in the distributed setting. We propose the *Strong Delayed Writes (SDW)* algorithm, enabling distributed sketch implementation while guaranteeing relaxed strongly linearizabile snapshots, by propagating updates network wide. Another approach is constructing sketches separately, and periodically reporting the measurements to a centralized server. Traditionally, nodes symmetrically compress their summaries. We show that communication can be reduced by considering the amount of traffic observed by each node and compressing accordingly. We illustrate this approach for three common sketches and perform extensive simulations to show that our sketches send smaller summaries than traditional ones while retaining similar error bounds.

Notation and Abbreviations

CM	CountMin (Sketch)
DDoS	Distributed Denial of Service
HLL	Hyper-Log-Log (Sketch)
IVL	Intermediate Value Linearizability
KMV	K-Minimum-Values (Sketch)
PAC	Probably-Approximately-Correct
SWMR	Single-Writer-Multi-Reader (Register)

Chapter 1

Introduction

Being able to process, in real-time, continuous high-volume *streams* of data is a common requirement in multiple analytical platforms. Examples of such applications are exploratory data analysis [116], operation monitoring [1], data aggregation [4, 19], graph mining [26] and more.

Unlike conventional database query processing, which requires several passes over static archived data, data-stream processing algorithms often rely on building concise approximate (yet, accurate) *sketches* [27] of the input stream in real-time. A sketch data structure is essentially a succinct (sublinear) summary of a stream that approximates a specific query, for instance, unique element count [19], quantile values [4], or frequent items [31].

Sketches typically provide a tradeoff between accuracy and space, and generally come with a closed-form accuracy analysis in a sequential setting. Such sketches are called *probably approximately correct (PAC)*, providing an approximation within some error ϵn with a failure probability bounded by some parameter δ, for sketch parameters (ϵ, δ). Practical sketch implementations have recently emerged in toolkits [11] and data analytics platforms (e.g., PowerDrill [61], Druid [39], Hillview [62], and Presto [67]).

1.1 Background: Sequential Sketches

A sketch S summarizes a collection of elements $\{a_1, a_2, \dots, a_n\}$, processed in some order given as a stream $A = a_1, a_2, \dots, a_n$. The desired summary is agnostic to the processing order, but the underlying data structures may differ due to the order. An important property of the sketches we consider in this thesis is *mergeability* : Two sketches with parameters (ϵ, δ) can be merged into a single sketch with the same parameters. We capitalize on this property, which enables computing a sketch over the whole stream by merging sketches over substreams, which has also been exploited in previous work [111, 99, 32].

The sketch's API is:

S.**init()** initializes S to summarize 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; the optional argument *arg* may, for example, be the requested quantile.

S.**merge**(S') merges sketches S and S' into S; i.e., if S initially summarized stream A and S' summarized A', then after this call, S summarizes the concatenation of the two, $A||A'$.

1.1.1 Θ Sketch

Computing the exact number of distinct elements, also called stream *cardinality* can be done with storage complexity essentially proportional to the stream size. Sketches estimating the stream cardinality have been widely studied [19, 49, 61, 50, 40], and provide estimates while using sublinear memory. All known efficient cardinality estimators rely on randomization, which is ensured by the use of *hash* functions, which hashes stream elements to the interval $[0, 1]$. Using this hash we can decrease memory usage. For example, we can maintain only elements whose hashes are less than $1/4$, and our estimate would be 4 times the count. However, the threshold still implies linear memory usage. Instead, the threshold should decrease as the number of unique elements increases. This is the idea behind the Θ sketch.

The Θ sketch is based on the *k Minimum Values (KMV)* algorithm [19] 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 [15]. The merge method adds a batch of samples to *sampleSet*.

1.1.2 CountMin Sketch

Estimating flow sizes is a required capability in many networking applications [43, 18, 90], in fields as diverse as accounting, monitoring, load balancing and filtering, and even

Algorithm 1 Θ sketch.

```
1:  variables
2:      sampleSet, init $k$ 1's                              ▷ samples
3:      $\Theta$, 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) $\geq \Theta$ then return
10:     add $h$(arg) to $sampleSet$
11:     keep $k$ smallest samples in $sampleSet$
12:     $\Theta \leftarrow max(sampleSet)$
13:     $est \leftarrow$ (|sampleSet| $-$ 1) / $\Theta$
14: procedure MERGE(S)
15:     sampleSet $\leftarrow$ merge sampleSet and $S$.sampleSet
16:     keep $k$ smallest values in sampleSet
17:     $\Theta \leftarrow max$(sampleSet)
18:     $est \leftarrow$ (|sampleSet| $-$ 1) / $\Theta$
```

beyond networking. Counting the exact size of every flow is often challenging due to a typically large number of active flows at a specific time, making it difficult to maintain a counter-per-flow within a memory accessible at the line rate. There can be two types of errors in the estimation of a flow size: Overestimations and underestimations. The state-of-the-art data structure for flow size estimation is the Count-Min (CM) sketch suggested by Cormode and Muthukrishnan in 2005 [31], which overestimates the flow sizes.

The CM sketch is instantiated with parameters ϵ and δ, where the flow size estimation is within error ϵ with probability at least $1-\delta$. It is comprised of a two dimensional array of counters of size $d \times w$, where $d = \lceil \ln 1/\delta \rceil$ and $w = \lceil e/\epsilon \rceil$, and all counters are initialized to 0. Note that the number of columns is determined by the error (and, conversely, the error is determined by the number of columns), and the probability is determined by the number of rows. A set of d hash functions is used to map a flow to d counters, one in each of its rows. Upon arrival of an element in the flow, each of these counters is incremented.

To estimate the size of a flow, its d selected counters are considered and the size is estimated as the minimum among these counters. Since multiple flows can contribute to the same counter, the computed value is potentially larger than the exact one. In case other flows contributed to all d counters then an overestimation occures.

The CM sketch is illustrated in Figure 1.1. Flows I, II of sizes 4, and 5, respectively, are recorded in the sketch (shown on the left side). Each flow increases the value of $d = 3$ counters by its size. The size of Flow III (right side) is estimated by querying the CM as the minimal among the d counters it is mapped to.

While, as mentioned, the CM can observe overestimations [31], the accuracy guarantees can be described as follows: When using CM with width w and depth d the

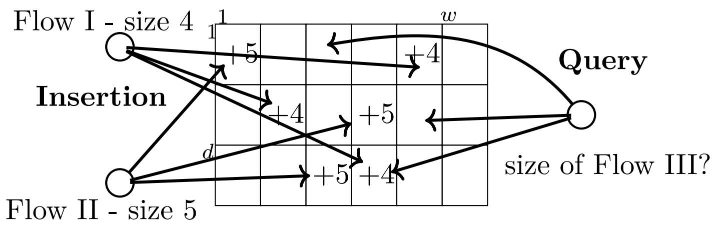


Figure 1.1: The Count-Min sketch (CM) [31], allowing flow size estimation. Each flow is mapped to a single counter in each row. Flow I is mapped to counter 5 in row 1, counter 2 in row 2, and counter 4 in row 3. The mappings are presented in the figure. A flow size is estimated as the minimum among the counters it is mapped to by a set of hash functions.

estimation $\hat{f}$ of flow f satisfies with probability $1-\delta$

$$\hat{f} \leq f + \epsilon N.$$

Here, N is the number of packets in the measured stream, $w = \lceil \frac{e}{\epsilon} \rceil$ (for Euler's number e) and $d = \left\lceil \ln \frac{1}{\delta} \right\rceil$.

1.1.3 Quantiles Sketch

Understanding the data distribution is a fundamental task in data management and analysis, used in applications such as exploratory data analysis [116], operations monitoring [1], and more. The Quantiles sketch family captures this task [85, 4, 51, 30]. The sketch represents the quantiles distribution in a stream of elements, such that for any $0 \leq \phi \leq 1$, a query for quantile ϕ returns an estimate of the $\lfloor n\phi \rfloor^{\text{th}}$ largest element in a stream of size n. For example, quantile $\phi = 0.5$ is the median. Due to the importance of quantiles approximation, Quantiles sketches are a part of many analytics platforms, e.g., Druid [38], Hillview [23], Presto [95], and Spark [110].

The Quantiles sketch proposed by Agarwal et al. [4] consists of a hierarchy of arrays, where each array summarizes a subset of the overall stream. The sketch is instantiated with a parameter k, which is a function of (ϵ, δ). The first array, denoted level 0, consists of at most $2k$ elements, and every subsequent array, in levels $1, 2, \ldots,$ consists of either 0 or k elements at any given time.

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

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

1.1.4 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 $1/\sqrt{k-2}$ [19].

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 $1 - \delta$ [4].

1.2 Concurrent Sketches - Our Results

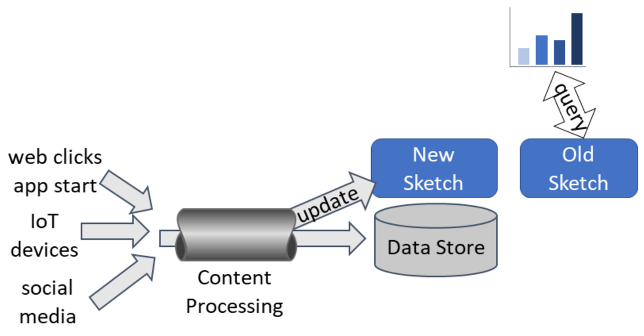


Figure 1.2: Using sketches in epochs.

Practical sketch implementations are provided within toolkits [11]. However, as of the time when this work had begun implementations were not thread-safe [13], allowing neither parallel data ingestion nor concurrent queries and updates; concurrent use was prone to exceptions and gross estimation errors. Applications using these libraries were therefore required to explicitly protect all sketch API calls by locks [108, 13]. 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 1.2. This practice leads to stale query results and thus loses the real-time quality of the system.

In Chapter 2 we explore a framework that takes advantage of the mergeability property [27], which enables computing a sketch over a stream by merging sketches over sub-streams. We provide a framework that is able to simultaneously provide queries and updates to the same sketch from multiple threads. In contract to previous methods (e.g., [61, 32]), which explore distributed sketching, our method is based on shared memory and constantly propagates results to a queryable sketch.

We then instantiate the framework with the Θ sketch described in Section 1.1.1 and analyze its performance. We find that we achieve a speedup of $10x$ with 12 threads, as, after some elements are processed, most elements are hashed and discarded at the thread level, and very few are propagated to the shared sketch. We also contributed our solution to open-source. We then plugged in two other sketches and identify two performance bottlenecks, which we address next.

The first performance issue is that our query requires an *atomic snapshot*. For some sketches (e.g., KMV [19]) we show that such a snapshot is simple to implement and does not effect performance. For other sketches (e.g., CountMin [31]) such a snapshot may be costly to implement. Secondly, we still require merging from multiple threads. Some sketches (e.g., CountMin [31]) have a quick merge operation. Others (e.g., Quantiles [4]) require a lengthy merge operation which results in limited scalability due to the merge being a sequential bottleneck.

To address the first issue, we first revisit some important questions when parallelzing sketches, for instance: *What are the semantics of overlapping operations in a concurrent sketch?*, *How can we prove error guarantees for such a sketch?*, and, in particular, *Can we reuse the myriad of clever analyses of existing sketches' error bounds in parallel settings without opening the black box?*

The framework presented in Chapter 2 uses *strong linearizablity* [53] as its correctness semantics. But satisfying such semantics require querying the sketch via an atomic snapshot. Roughly speaking, linearizability requires each parallel execution to have a linearization, which "looks like" some sequential execution. But sometimes linearizability is too restrictive leading to a high implementation cost, as has motivated other works on relaxing linearizability [3, 59, 6, 8], and is also shown herein.

In Chapter 3 we propose *Intermediate Value Linearizability (IVL)*, a new correctness criterion for concurrency objects that return a quantitative value. Intuitively, the return value of an operation of an IVL object is bounded between two legal values that can be returned in linearizations. For example, consider incrementing a counter from 7 to 10. Under linearizability, a read that occurs concurrently with the update may return either a 7 or a 10, while under IVL, the read may return any number between 7 and 10. The motivation for allowing this is that if the system designer is happy with either of the legal values, then the intermediate value should also be fine.

We show that concurrent IVL implementations of sequential sketches inherit the error bounds from their sequential counterparts. The importance of this is that it *provides a generic way to leverage the vast literature on sequential sketches [89, 48, 25, 82, 31, 4] in concurrent implementations.* We then present a concurrent IVL implementation of the CountMin sketch, and, by the semantics, deduce that the concurrent sketch adheres to the error guarantees of the original sequential one, without having to "open" the analysis. As IVL is less restrictive than linearizability, an IVL snapshot can be used instead of the atomic one to circumvent the performance degradation induced by the atomic snapshot.

The second performance bottleneck is the merge operation. Addressing this requires a concurrent execution of the merge operation. To this end, we propose Quancurrent, a highly scalable concurrent Quantiles sketch. Like the framework presented in Chapter 2, Quancurrent relies on local buffering of stream elements, which are then propagated in bulk to a shared sketch. But Quancurrent improves on the framework by eliminating the latter's sequential propagation bottleneck, which, in the Quantiles sketch, mostly stems from the need to sort large buffers. In a nutshell, Quancurrent leverages unsynchronized concurrent propagation, which introduces a small sampling error, but increases speedup. Furthermore, instead of working only with thread-local and global buffers, Quancurrent uses also intermediate NUMA-local buffers that are propagated concurrently to a shared state, which increses the speedup by reducing the sequential bottleneck. The multi-level concurrent propagation allows Quancurrent to work with much smaller buffers, thus increasing the query freshness by reducing the number of updates which can be "missed" by a query. Quancurrent is presently in Chapter 4.

1.3 Distributed Sketches - Our Results

Large-scale stream processing applications are inherently *distributed*, with several remote sites observing their local stream(s) and exchanging information through a communication network. Analyzing a single site does not always give the fullest picture. For example, an attacker may be executing a distribute denial of service (DDoS) attack on the network as a whole, but maintaining a low profile on each site individually. Only the aggregation of the data shows the attack. To this end, we model the input to the network as a single stream, with disjoint parts of the stream sampled by multiple *ingestion nodes*. The ingestion nodes periodically propagate their local sketches to a *central node*, as illustrated in Figure 1.3. The network has to handle the trade-off of sending data packets vs. sending crucial control packets with sketch information [124, 20]. A way to reduce these control packets is by compressing the sketches before transferring them to a central analytics node.

In Chapter 5, we develop a compression method named CM-SKTC for a single CountMin sketch, which allows general compression ratios, and then present the Traffic-Aware CM sketch, denoted TA-CM. We also present Traffic-Aware K-minimum-values (KMV), denoted TA-KMV – a traffic-aware compression ratio for nodes implementing

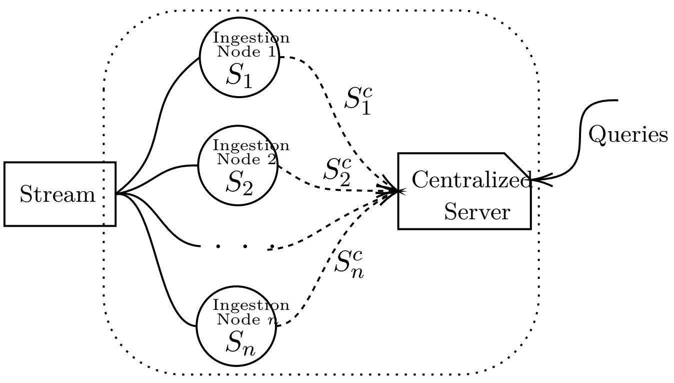


Figure 1.3: Network-wide measurement over n ingestion nodes: Queries are answered by a centralized server collecting summaries $S_1^c, \ldots, S_n^c$ of the local sketches $S_1, \ldots, S_n$ (e.g., the Count-Min sketch (CM) or the K-minimum-values (KMV) sketch) in the ingestion nodes.

distributed distinct flow count with the KMV sketch. Finally, we present Traffic-Aware HyperLogLog (HLL), denoted TA-HLL – a traffic-aware compression ratio for nodes implementing distributed distinct flow count with the HLL sketch. We then implement our solutions and evaluate them on both the CAIDA [114] and MAWI [100] datasets, and show similar average relative error with up to 10% fewer control packets in the case of the CM sketch, but in the case of TA-KMV and TA-HLL we show similar average relative error with a reduction of 50% in conrol packets.

Finally, in Chapter 6, we present an algorithm for constructing a distributed shared sketch, which supports local reads and writes, ensuring relaxed strong linearizability, similarly to the concurrent sketches presented in Section 1.2. The algorithm batches updates into *windows*, and synchronizes window advancement. To distribute the sketch, each node maintains three objects holding copies of the sketch, R_u, R_r, and R_s. At any given time R_u is updated, R_r is queried, and R_s is synchronized across the network. The sketches' roles are alternated in a round-robin manner on window advancement. In [121] we show that such an object can be efficiently implemented on programmable switches only using the data-plane.

1.4 Conclusions

Data sketches are an indispensable tool in big data analysis, as they produce results orders-of magnitude faster than exact solutions, and with mathematically proven error bounds. Designing a system around sketches allows simplification of the system's architecture and a reduction in overall compute resources required for the difficult computational tasks of analysis. As such, they are included in multiple practical toolkits. Yet virtually all prior work does not consider distribution or concurrency, but in the age of the multi-core and cloud computing this is a requirement. In this thesis we consider parallelism and data distribution as first class citizens in sketch design. Whereas naive parallelization may introduce a large error, we develop various examples of sketch designs that leverage parallelization for performance or deal with distributed data without introduction excessive error.

Chapter 2

Fast Concurrent Data Sketches

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. In this chapter, we present a generic approach to parallelizing data sketches efficiently and allowing them to be queried in real time, while bounding the error that such parallelism introduces. Utilizing relaxed semantics and the notion of strong linearizability we prove our algorithm's correctness and analyze 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.

2.1 Introduction

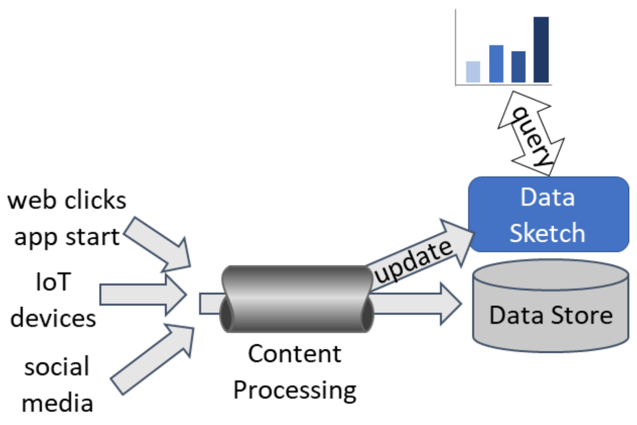


Figure 2.1: Stream processing pipeline with data sketch.

As previously mentioned, practical sketch implementations have recently emerged in toolkits [11] and data analytics platforms (e.g., PowerDrill [61], Druid [39], Hillview [62], and Presto [67]). 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 [108, 13]. 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 1.2 in the previous chapter. This practice leads to stale query results and thus loses the real-time quality of the system. Instead, we aim to query fresh data, as illustrated in Figure 2.1.

Our approach. We present a generic approach to parallelizing 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 synchronization. This is particularly challenging because sketch libraries are extremely fast, often processing tens of millions of updates per second.

We capitalize on the well-known sketch *mergeability* property [27], 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., [61, 32]), 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 2.2. Multiple worker thread buffer updates in local sketches and periodically merge them into a global sketch; queries access the latter.

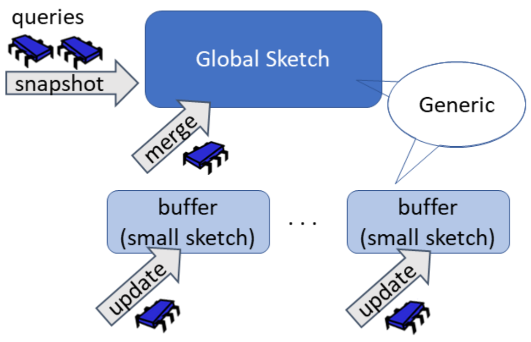


Figure 2.2: Concurrent sketches architecture.

We adaptively parallelize 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 [19], which estimates the number of unique elements in a stream; a popular sketch from the open-source

Apache DataSketches library [11]. We have contributed our implementation back to the Apache DataSketches library [34]. 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 (Section 2.4).

Figure 2.3 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.

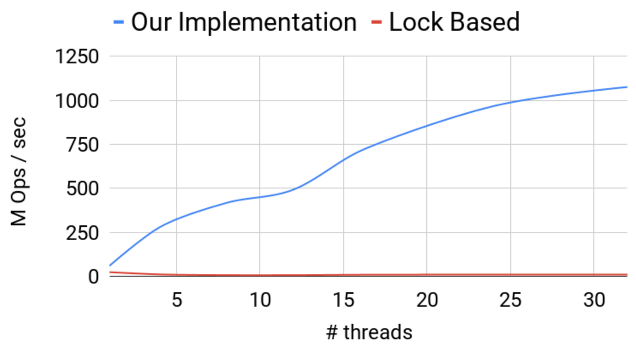


Figure 2.3: Scalability of DataSketches' Θ sketch protected by a lock vs. our concurrent implementation.

Error analysis. Concurrency might induce an error, and one of the main challenges we address is analyzing 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 flavor of *relaxed consistency* similar to [59, 6, 112] 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 [59] and priority queues [7, 98], we believe that they are a natural fit for data sketches. This is because sketches are typically used to summarize 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 randomized. We therefore first de-randomize the sketch's behavior 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 randomized algorithms, it is not enough to prove its linearizability. Rather, we prove that our generic concurrent algorithm instantiated with sequential sketch S satisfies *strong linearizability* [53] with regards to a relaxed sequential specification of the de-randomized S. We note, however, that supporting strong linearizability did not incur additional costs nor did it impact the relaxation; we were able to prove that our original design was strongly linearizable.

We then analyze the error for two types of relaxed sketches under random coin flips, with an adversarial scheduler that may delay operations in a way that maximizes the error. First, we consider the Θ sketch. For this sketch, its relative standard error has been analyzed, 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.

Section 2.2 lays out the model for our work. In Section 2.3 we formulate a flavor of relaxed semantics appropriate for data sketches. Section 2.4 presents our generic algorithm, and Section 2.5 proves strong linearizability of our generic algorithm. Section 2.6 analyses error bounds for example sketches. Section 2.7 empirically studies the Θ sketch's performance and error with different stream sizes. Finally, Section 2.8 concludes.

2.2 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 [71] and C++ [21]. 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 *linearization* 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. [53] have shown that in order to ensure correct behavior of randomized algorithms under concurrency, one has to prove *strong linearizability*:

Definition 2.2.1 (Strong linearizability)**.** 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 linearizable implementation of an object o if there is a prefix preserving function f that maps every execution σ of A to a linearization H of σ.

For example, executions of atomic variables are strongly linearizable.

2.3 Relaxed consistency for concurrent sketches

Previous work by Alistarh et al. [6] has presented a formalization for a randomized relaxation of an object. The main idea is to have the parallel execution approximately simulate the object's correct sequential behavior, 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 parallelize (namely the sketch) is itself randomized. Thus, there are two sources of error: (1) the approximation error in the sequential sketch and (2) the additional error induced by the parallelization. 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 derandomize the sequential sketch by delegating its coin flips to an oracle, and then analyze 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 [59] *out-of-order* relaxation, which generalizes quasi-linearizabilty [3]. 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 2.3.1 (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 behaviors. 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 derandomized 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 2.3.2 (r-relaxation)**.** The *r-relaxation* of SeqSketch is the set of histories that have r-relaxations in SeqSketch:

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

Note that our formalism slightly differs from that of [59] in that we start with a serialization 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.

An example is given in Figure 2.4, 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

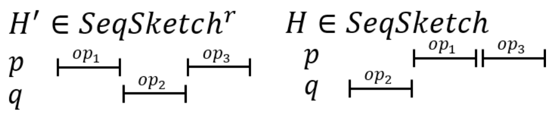


Figure 2.4: H is a 1-relaxation of H'.

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.

2.4 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 2.4.1, 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 serializes ingestion in small streams and parallelizes it in large ones. For clarity of presentation, we present in Section 2.4.2 the parallel phase of the algorithm, which provides relaxed semantics appropriate for large streams. Section 2.4.3 then discusses the adaptation for small streams.

2.4.1 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 linearizable. Our Θ sketch, shown below, simply accesses an atomic variable that holds the query result. In other sketches, for instance, CountMin [31], HyperLogLog [49, 61, 39, 67], and Quantiles [73], 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.

Pre-filtering. When multiple sketches are used in a multi-threaded algorithm, we can optimize 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 [117]. To support this optimization, 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) *summarizes* 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 $a_1, \dots, a_n$, namely after a sequential execution with the history

$$H = S.\mathit{update}(a_1), S.\mathit{resp}, \dots S.\mathit{update}(a_n), S.\mathit{resp}.$$

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

Given a sketch state s' that summarizes a stream A, if shouldAdd($S.\mathit{calcHint}()$, a) returns false then for every streams B_1, B_2 and sketch state s' that summarizes $A||B_1||a||B_2$, s' also summarizes $A||B_1||B_2$. 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 minimize 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*.

2.4.2 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

Algorithm 2 Additional methods for composable Θ sketch.

```
1: procedure SNAPSHOT
2:     $localCopy \leftarrow emptysketch$
3:     $localCopy.est \leftarrow est$
4:     return $localCopy$
5: procedure CALCHINT
6:     return $\Theta$
7: procedure SHOULDADD(H, arg)
8:     return $h(\text{arg}) < H$
```

the presentation, we first discuss an unoptimized version of our generic concurrent algorithm, (left column in of Algorithm 3), called *ParSketch*, and later an optimized 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 minimized. 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 synchronizes 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 synchronization in order to pass the hint.

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

Query threads use the snapshot method, which can be safely run concurrently with merge, hence there is no need to synchronize between the query threads and t_0. The freshness of the query is governed by the r-relaxation. In Section 2.5.2 we prove Lemma 2.4.1 below, asserting that the relaxation is Nb. This may seem straightforward

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 $t_i$ , $0 \leq i \leq N$
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:             $prop_i \leftarrow globalS.calcHint()$
116: procedure QUERY(arg)
117:     $localCopy \leftarrow globalS.snapshot(localCopy)$
118:     return $localCopy.query(arg)$
119: procedure UPDATE$_i(a)$
120:     if ¬shouldAdd($hint_i$, $a$) then return
121:     $counter_i \leftarrow counter_i + 1$
122:     $localS_i.update(a)$
123:     if $counter_i = b$ then
124:         $prop_i \leftarrow 0$
125:         wait until $prop_i \neq 0$
126:
127:         $hint_i \leftarrow prop_i$
128:         $counter_i \leftarrow 0$
129:
```

```
Optimised algorithm
201: variables
202:     composable sketch $globalS$, init empty
203:     constant $b$                          ▷ relaxation is $2Nb$
204:     for each update thread $t_i$ , $0 \leq i \leq N$
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:             $globalS.merge(localS_i[1\text{-}cur_i])$
214:             $localS_i[1 - cur_i]$ ←empty sketch
215:             $prop_i \leftarrow globalS.calcHint()$
216: procedure QUERY(arg)
217:     $localCopy \leftarrow globalS.snapshot(localCopy)$
218:     return $localCopy.query(arg)$
219: procedure UPDATE$_i(a)$
220:     if ¬shouldAdd($hint_i$, $a$) then return
221:     $counter_i \leftarrow counter_i + 1$
222:     $localS_i[cur_i].update(a)$
223:     if $counter_i = b$ then
224:
225:         wait until $prop_i \neq 0$
226:         $cur_i \leftarrow 1 - cur_i$
227:         $hint_i \leftarrow prop_i$
228:         $counter_i \leftarrow 0$
229:         $prop_i \leftarrow 0$
```

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 2.4.1. *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 optimized *OptParSketch* algorithm, which improves thread utilization 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 optimization 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 2.5.3 we prove the correctness of the optimized algorithm by simulating N threads of *OptParSketch* using $2N$ threads running *ParSketch*. We do this by showing a *simulation relation* [83]. We use forward simulation (with no prophecy variables), ensuring strong linearizability. We conclude the following theorem:

Theorem 1. OptParSketch *instantiated with SeqSketch is strongly linearizable with regards to* SeqSketch^r*, where $r = 2Nb$.*

2.4.3 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 initializing 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.

2.5 Proofs

In Section 2.5.1 we introduce some formalisms. In Section 2.5.2 we prove that the unoptimized algorithm is strongly linearizable with respect to the relaxed specification *SeqSketch*^{r} with $r = Nb$. Finally, in Section 2.5.3 we show that the the optimized algorithm is strongly linearizable with respect to the relaxed specification *SeqSketch*^{r} with $r = 2Nb$.

2.5.1 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 linearizable. By slight abuse of terminology, we refer to these operations as atomic steps, for example, we refer to the linearization 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 synchronize), 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 a sketch's state *summarizes* 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 "summarizes" to apply also to threads.

Definition 2.5.1 (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 *summarizes* the history summarized by $localS_i$ at time t. Otherwise, thread t_i summarizes the empty history at time t. The propagator thread t_0 summarizes 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 analyze each thread's steps in an execution, we first define the projection from execution σ onto a thread t_i.

Definition 2.5.2 (Projection)**.** Given a finite execution σ and a thread t_i, $\sigma\Big|_{t_i}$ is the subsequence of σ consisting of steps taken by t_i.

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

Definition 2.5.3 (Unpropogated updates)**.** Given a finite execution σ, we denote by $\mathrm{suff}_i(\sigma)$ the suffix of $\sigma\Big|_{t_i}$ starting at the last $globalS$.merge($localS_i$) event, or the beginning of σ if no such event exists. The unpropogated suffix $\mathrm{up\_suff}_i(\sigma)$ of update thread i is the subsequence of $\mathcal{H}(\mathrm{suff}_i(\sigma))$ consisting of $update(a)$ executions in $\mathrm{suff}_i(\sigma)$ for which shouldAdd($hint_i, arg$) returns true in line 120.

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

Definition 2.5.4. Given a finite sequential history H, $\mathcal{S}(H)$ is the stream $a_1, \dots, a_n$ 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 2.5.5. Given a finite sequential history H and two method invocations M_1, M_2 in H, we denote $M_1 \prec_H M_2$ if M_1 precedes M_2 in H.

2.5.2 Proof of unoptimized algorithm

Our strong linearizability 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 (linearization 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 *linearization points* defined as follows:

- An updates' linearization point is its invocation
- A query's linearization 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 $i = 1, \dots, N$, 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 initialized 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 2.5.6. *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 2.5.7. *At any point 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 summarizes 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 summarizes 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 summarized 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$_i$* summarizes up_suff$_i(\sigma')$, and so after the update, *localS$_i$* summarizes up_suff$_i(\sigma')$, *update*(*arg*) = up_suff$_i(\sigma)$. From Invariant 1 *prop$_i$* $\neq 0$, therefore, by Definition 2.5.1, t_i summarizes 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 summarizes the history up_suff$_i(\sigma')$. Because before the step *prop$_i$* $\neq 0$, *localS$_i$* summarizes 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 summarizes 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 2.5.1, t_i summarizes 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 2.5.3, after this step up_suff$_i(\sigma)$ is empty. As this step is a *merge*, *merge_ctr*[i] is increased by one, so $sig\_ctr[i] = merge\_ctr[i]$ by Observation 2.5.7. Therefore, by Definition 2.5.1, t_i summarizes 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 2.5.1, $localS_i$ summarizes 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 2.5.1, t_i summarizes the same history as $localS_i$, i.e., the empty history. Therefore, t_i summarizes up_suff$_i(\sigma)$, preserving the invariant.

□

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

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

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 summarizes $f(\sigma')$. And by Invariant 2, before the update, t_i summarizes up_suff$_i(\sigma')$, and by Invariant 1 $localS_i$ summarizes the same history. Let $A = \mathcal{S}(f(\sigma))$, and $B = \mathcal{S}(\text{up\_suff}_i(\sigma'))$. After the merge *globalS* summarizes $A||B$. Therefore, t_0 summarizes $f(\sigma)$ 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* summarized $f(\sigma'')$. 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* summarizes $B = A||B_1$. By the assumption of *shouldAdd*, if shouldAdd(H, arg) returns false, then if a sketch summarizes $B = A||B_1||B_2$, then it also summarizes $B = A||B_1||a||B_2$. Let $B_2 = \emptyset$, then *globalS* summarizes $B = A||B_1||B_2$, therefore also summarizes $A||B_1||a||B_2 = A||B_1||a$. Therefore, after the step, *globalS* summarizes $f(\sigma)$ 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 2.5.8 (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 summarizes $f(\sigma')$ at point v, therefore *globalS* summarizes $f(\sigma')$ at the same point, therefore *globalS* summarizes 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* summarizes 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 2.5.9. *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 2.5.10. *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 linearization point. As Q's visibility point and linearization point are equal, it follows that if $Q \prec_{l(\sigma)} O$ then $Q \prec_{f(\sigma)} O$. □

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

Lemma 2.5.11. *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 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 2.5.12. *Given a finite execution σ of ParSketch, $|f(\sigma)| \geq |l(\sigma)| - Nb$.*

Proof. From Lemma 2.5.11, $|\text{up\_suff}_i(\sigma)| \leq b$. 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 2.5.13. *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 2.5.10, 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 2.5.12, $|f(\sigma^{pre})| \geq |l(\sigma^{pre})| - Nb$. 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 2.3.2, $f(\sigma)$ is an Nb-relaxation of $l(\sigma)$. □

Putting together Lemma 2.5.9 and Lemma 2.5.13, 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 2.4.1.

2.5.3 Optimized algorithm proof

We denote the optimized 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 [83], 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 linearizable with regards to $SeqSketch^{2Nb}$ [47, 17].

Consider an arbitrary worker thread t_i for the optimized 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 preprocessing, so we can simulate each thread by two threads. For example in Figure 2.5, 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 $A = A_1||A_2||\ldots||A_n$ and we assume n is even. Each A_i is evaluated against the *hint* written above it. The thread t_i^0 simulates processing $A_1||A_3||\ldots||A_{n-1}$, and thread t_i^1 simulates processing $A_2||A_4||\ldots||A_n$.

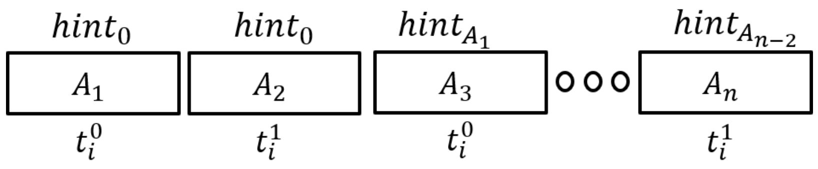


Figure 2.5: Simulation of processing $A = A_1||A_2||\ldots||A_n$.

The simulation uses auxiliary variables oldHint$_i^0$, and oldHint$_i^1$, both initialized 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 $auxCount_i$ initialized 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 initialized 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 2.6 shows a mapping when cur_i equals 0, before executing line 227. Table 2.1 shows the steps taken by t_i^0 and t_i^1 when $cur_i = 0$ before line 223.

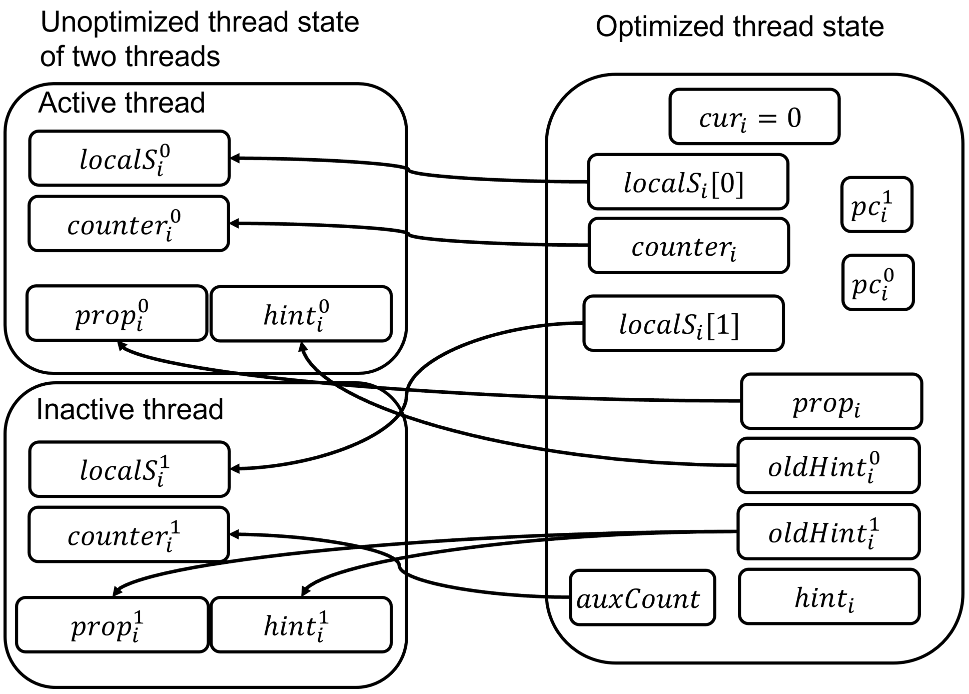


Figure 2.6: Reference mapping of g when cur_i equals 0 before executing line 227.

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 2.1: 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.

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 2.5.14. *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 linearizability [17], intuitively, because the mapping of all ParSketch's steps – including linearization 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 linearizable with regards to* SeqSketch^r*, where* $r = 2Nb$*.*

2.6 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 anlayzes of existing sketches. In Section 2.6.1, we consider the relative standard error of the Θ sketch, which was used in the original analysis of the sketch. In Section 2.6.2 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.

2.6.1 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 optimized 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 summarized in Table 2.2. Our analysis relies on known results from order statistics [35]. 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 2.2: Expectation and RSE of Θ sketch with numerical values for $r = 8, k = 2^{10}, n = 2^{15}$.

We would like to analyze 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 analyze 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 [19], i.e., $E[e] = n$ the number of unique elements. Moreover, previous work [115] has analyzed 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 maximize 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 2.6.1. *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 maximizes $|\frac{k-1}{x} - n|$ for a given n is either $M_{(0)}$ or $M_{(j)}$.*

Proof. Assume for the sake of contradiction that the variable that maximizes $|\frac{k-1}{x} - n|$ 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.

$\square$

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,

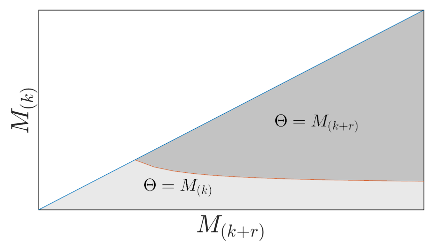


Figure 2.7: Areas of $M_{(k)}$ and $M_{(k+r)}$. In the dark gray $\mathcal{A}_s$ induces $\Theta = M_{(k+r)}$, and in the light gray, $\Theta = M_{(k)}$. The white area is not feasible.

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 2.6.2. *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.

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

□

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 maximizes the error:

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

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:

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

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

$$\begin{aligned} E[e_{\mathcal{A}_s}] &= \int_0^1 \int_0^{m_{k+r}} e_{\mathcal{A}_s} \cdot f_{M_{(k)},M_{(k+r)}}(m_k, m_{k+r})\, dm_k\, dm_{k+r} \\ E[e^2_{\mathcal{A}_s}] &= \int_0^1 \int_0^{m_{k+r}} [e_{\mathcal{A}_s}]^2 \cdot f_{M_{(k)},M_{(k+r)}}(m_k, m_{k+r})\, dm_k\, dm_{k+r} \end{aligned} \tag{2.1}$$

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

$$\begin{aligned} \text{RSE}[e_{\mathcal{A}_s}]^2 &= \frac{1}{n^2} \int_0^1 \int_0^{m_{k+r}} (e_{\mathcal{A}_s} - n)^2 \cdot f_{M_{(k)},M_{(k+r)}}(m_k, m_{k+r})\, dm_k\, dm_{k+r} \\ &= \frac{1}{n^2} \int_0^1 \int_0^{m_{k+r}} (e_{\mathcal{A}_s} - E[e_{\mathcal{A}_s}] + E[e_{\mathcal{A}_s}] - n)^2 \cdot f_{M_{(k)},M_{(k+r)}}(m_k, m_{k+r})\, dm_k\, dm_{k+r} \\ &\le \frac{1}{n^2}\left(\sigma^2(e_{\mathcal{A}_s}) + (e_{\mathcal{A}_s} - n)^2\right) \\ \text{RSE}[e_{\mathcal{A}_s}] &\le \sqrt{\frac{\sigma^2(e_{\mathcal{A}_s}) + (e_{\mathcal{A}_s} - n)^2}{n^2}} \\ &\le \sqrt{\frac{\sigma^2(e_{\mathcal{A}_s})}{n^2}} + \sqrt{\frac{(e_{\mathcal{A}_s} - n)^2}{n^2}} \end{aligned} \tag{2.2}$$

In Figure 2.7 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 2.7 using their joint probability function from order statistics. Equations 2.1 and 2.2 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 2.2, are shown in Table 2.2.

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

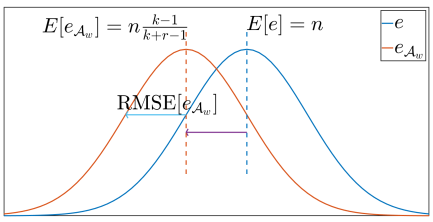


Figure 2.8: Distribution of estimators e and $e_{\mathcal{A}_w}$. The RSE of $e_{\mathcal{A}_w}$ with regards to n is bounded by the relative bias plus the RMSE of $e_{\mathcal{A}_w}$.

$n - n\frac{k-1}{k+j-1}$. Obviously this is maximized for $j = r$. The orange curve in Figure 2.8 depicts the distribution of $e_{\mathcal{A}_w}$, and the distribution of e is shown in blue.

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:

$$\sqrt{\frac{\sigma^2[e_{\mathcal{A}_w}]}{E[e_{\mathcal{A}_w}]}} < \frac{1}{k-2} \tag{2.3}$$

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 2.2:

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

Using Equation 2.3:

$$\text{RSE}[e_{\mathcal{A}_w}] < \sqrt{\frac{1}{k-2}} + \frac{r}{k-2} \tag{2.4}$$

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.

2.6.2 Error bounds for PAC sketches

We now provide a generic analysis, considering a PAC sketch as a black box. Section 2.6.2 studies quantiles sketches, and in Section 2.6.2, 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.

Quantiles error bounds

We now analyze 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., [4]) 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 maximize 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 $0 \le i + j \le r$. 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:

$$[(\phi - \epsilon)(n - (i + j)) + i, (\phi + \epsilon)(n - (i + j)) + i]\,. \tag{2.5}$$

This can be rewritten as:

$$\begin{aligned}&[\phi n - (\phi j - (1 - \phi)i + \epsilon(n - (i + j))),\\ &\phi n + ((1 - \phi)i - \phi j + \epsilon(n - (i + j)))]\end{aligned} \tag{2.6}$$

Note that this interval is symmetric around $\phi(n - (i + j)) + i$. The adversary attempts to maximize the distance of the edges of this interval from the true rank, (i.e., maximize 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 $0 \le i + j \le r$, we show that this expression is maximized for $i + j = r$.

Claim 2.6.3. *Given $0 \leq i, j$ such that $0 \leq i + j \leq r$, the expression $|(1 - \phi)i - (\phi)j|$ is maximized for $(i, j) = (x, y)$ such that $x + y = r$.*

Proof. Assume by contradiction that the expression given in the claim is maximized 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 maximized 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 2.6.4. *For $\phi \leq 0.5$ the adversary maximizes the distance by choosing $i = r$ (and therefore $j = 0$) and for $\phi > 0.5$ the adversary maximizes the error by choosing $i = 0$ (and therefore $j = r$).*

We begin by analyzing the range given in Equation 2.6 for $0 \leq \phi \leq 0.5$.

Claim 2.6.5. *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 $i \leq r$.

$$f(i, j) = (1 - \phi)i - \phi j + \epsilon(n - (i + j)) \leq (1 - \phi)i + \epsilon(n - i) \leq (1 - \phi - \epsilon)i + \epsilon n \tag{2.7}$$

$$\leq (1 - \phi - \epsilon)r + \epsilon n = (1 - \phi)r + \epsilon(n - r) = f(r, 0) \tag{2.8}$$

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

$$\phi j - (1 - \phi)i + \epsilon(n - (i + j)) \leq (1 - \phi)j + \epsilon(n - j) \leq (1 - \phi)r + \epsilon(n - r) \tag{2.9}$$

□

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

Claim 2.6.6. *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 $0 \le i + j \le r$, then $i \le r$.

$$f(i,j) = \phi i - (1-\phi)j + \epsilon(n-(i+j)) \le \phi i + \epsilon(n-i) \le (\phi-\epsilon)i + \epsilon n \le \phi r + \epsilon(n-r) = f(r,0) \tag{2.10}$$

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

$$(1-\phi)i - \phi j + \epsilon(n-(i+j)) \le \phi i + \epsilon(n-i) \le \phi r + \epsilon(n-r) \tag{2.11}$$

□

Putting the two claims together we get:

Claim 2.6.7. *For $0 \le \phi \le 1$ and $i, j > 0$ such that $0 \le i+j \le r$ and $\epsilon \ll 0.5$, then: (1) $\phi i-(1-\phi)j+\epsilon(n-(i+j)) \le r+\epsilon(n-r)$, and (2) $(1-\phi)i-\phi j+\epsilon(n-(i+j)) \le r+\epsilon(n-r)$.*

Proof. From Claim 2.6.5, for $0 \le \phi \le 0.5$ then both inequalities are bounded by $(1-\phi)r + \epsilon(n-r)$, and as $\phi \ge 0$ then $(1-\phi)r + \epsilon(n-r) \le r + \epsilon(n-r)$.

From Claim 2.6.6, for $0.5 < \phi \le 1$ then both inequalities are bounded by $\phi r + \epsilon(n-r)$, and as $\phi \le 1$ then $\phi r + \epsilon(n-r) \le r + \epsilon(n-r)$. □

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

Lemma 2.6.8. *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 $0 \le i + j \le r$, the rank of the element returned by the query is in the range given in Equation 2.6 w.p. at least $1 - \delta$:

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

From Claim 2.6.7, 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.

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 maximizes 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.

2.7 Θ sketch evaluation

This section presents an evaluation of an implementation of our algorithm for the Θ sketch. Section 2.7.1 presents the methodology for the analysis. Section 2.7.2 shows the results under different workloads and scenarios. Finally, Section 2.7.3 discusses the tradeoff between accuracy and throughput.

2.7.1 Setup and methodology

Our implementation [34] extends the code in Apache DataSketches [11], 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 2.4. 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 synchronization.

Unless stated otherwise, we use k=4096, which is commonly used [11] 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 2.4 with $r = 2Nb$. Recall that the analysis in Section 2.6.1 (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 minimize 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 linearization; because we cannot empirically deduce the linearization 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.

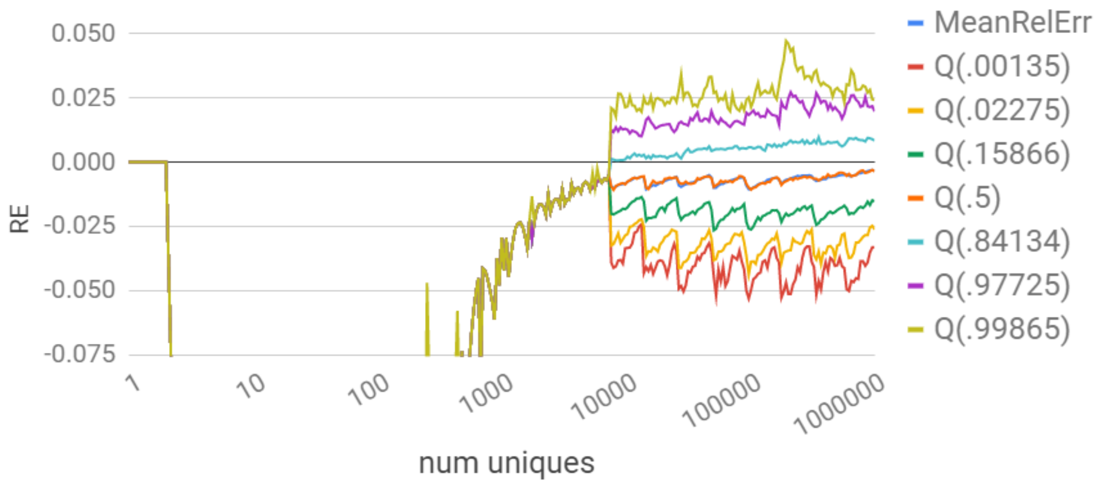
As in the performance evaluations, the x-axis represents the number of unique values fed into the sketch by a single writing thread. For each size x, one trial logs the estimation result after feeding x unique values to the sketch. In addition, it logs the Relative Error (RE) of the estimate, where $RE = MeasuredValue/TrueValue - 1$. 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".

2.7.2 Results

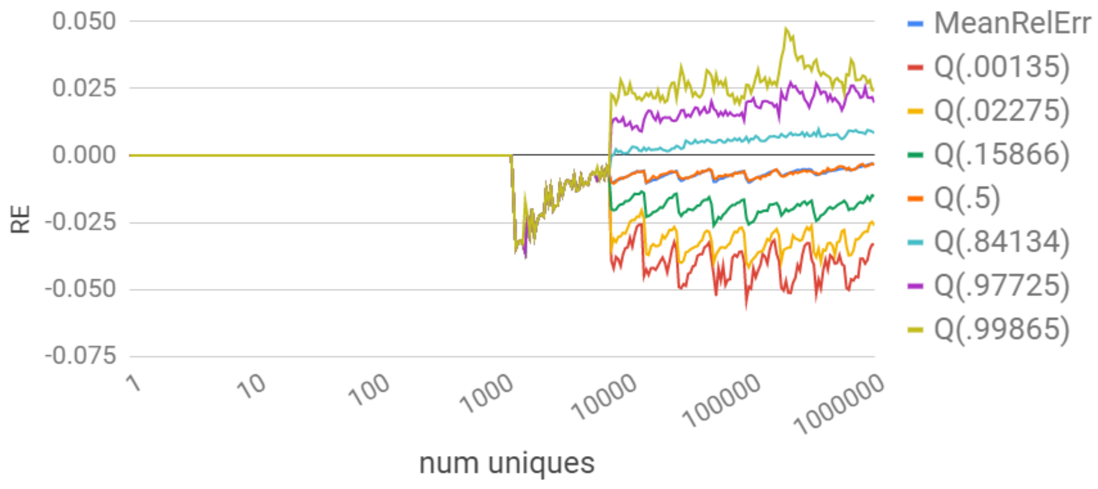
Accuracy results Our first set of tests runs on a 12-core Intel Xeon E5-2620 machine. The accuracy results for the concurrent Θ sketch without eager propagation are presented in Figure 2.9a. There are two interesting phenomena worth noting. First, it is interesting to see empirical evaluation reflecting the theoretical analysis presented in Section 2.6.1, where the pitchfork is distorted towards underestimating the number of unique values. Specifically, the mean relative error is smaller than 0 (showing a tendency towards underestimating), and the relative error in all measured quantiles tends to be smaller than the relative error of the sequential implementation.

Second, when the stream size is less than $2k$, $\Theta = 1$ and the estimation is the number of values propagated to the global sketch. If we forgo eager propagation, the number of values in the global sketch depends on the delay in propagation. The smaller the sketch, the more significant the impact of the delay, and the mean error reaches as high as 94% (the error in the figure is capped at 10%). As the number of propagated values approaches $2k$, the delay in propagation is less significant, and the mean error decreases. This excessive error is remedied by the eager propagation mechanism. The maximum error allowed by the system is passed as a parameter to the concurrent sketch, and the global sketch uses eager propagation to stay within the allowed error limit. Figure 2.9b 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 2.10a presents throughput measurements for a write-only workload. The results are shown in $\log\log$ scale. Figure 2.10b zooms-in on the throughput of large streams. As explained in Section 2.7.1, we compare the concurrent implementation to a lock-based approach. The number of threads in both implemen-



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



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

Figure 2.9: Concurrent Θ measured quantiles vs RE, $k = 4096$.

tations 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 2.11 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 2.3 (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.

2.7.3 Accuracy-throughput tradeoff

The speedup achieved by eager propagation in small streams is presented in Figure 2.12. This is an additional advantage of eager propagation in small streams, beyond the accuracy benefit reported in Figure 2.9. 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.

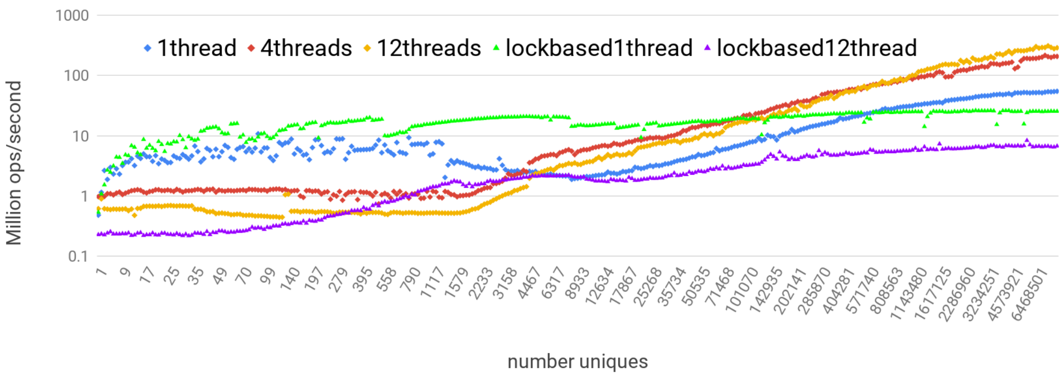
Next we discuss the impact of k. One way to increase the throughput of the concurrent Θ sketch is by increasing the size of the global sketch, namely increasing k. On the other hand, this change also increases the error of the estimate. Table 2.3 presents the tradeoffs between performance and accuracy. 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

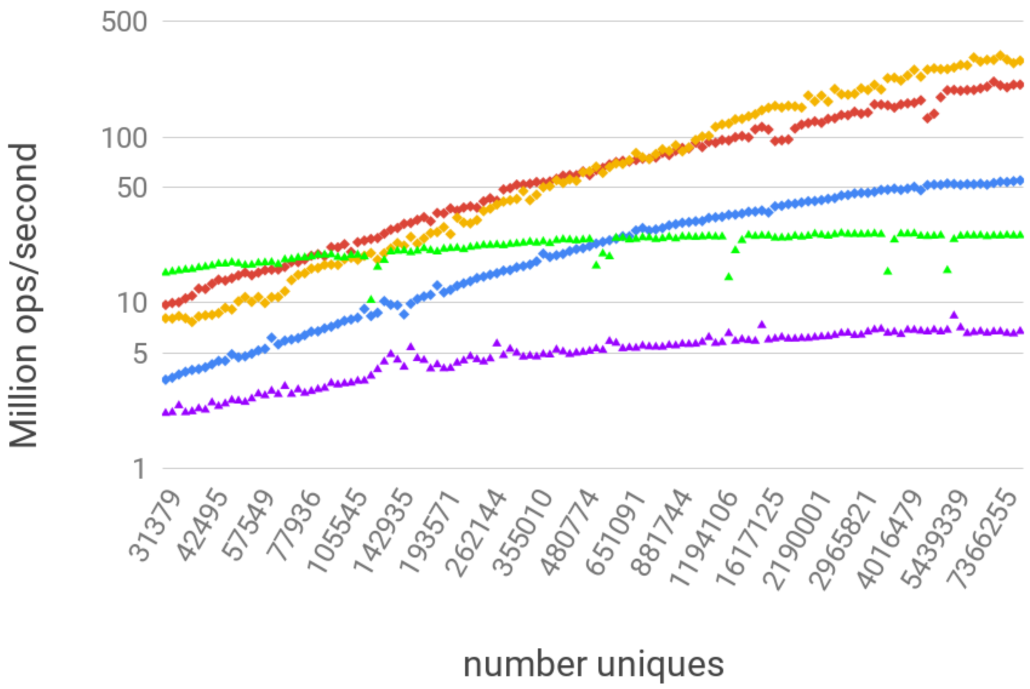
Table 2.3: Performance vs accuracy as a function of k.

2.8 Conclusions

Sketches are widely used by a range of applications to process massive data streams and answer queries about them. Library functions producing sketches are optimized to be extremely fast, often digesting tens of millions of stream elements per second. We presented a generic algorithm for parallelizing such sketches and serving queries in real-time; the algorithm is strongly linearizable 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 analyzed 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.



(a) Throughput, loglog scale



(b) Zooming-in on large sketches

Figure 2.10: Write-only workload, $k = 4096$, $\epsilon = 0.04$.

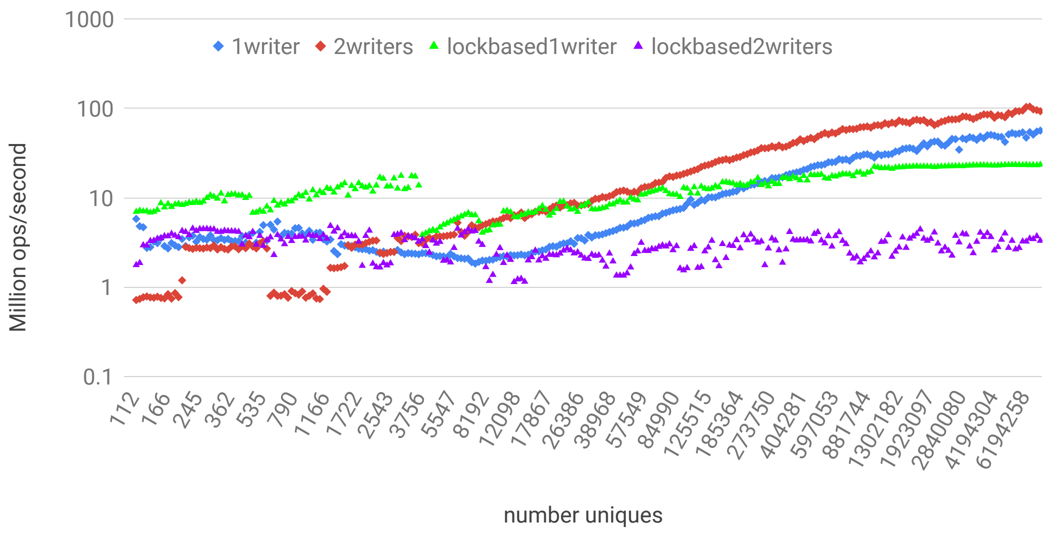


Figure 2.11: Mixed workloads: writers with background reads, $k = 4096$, $\epsilon = 0.04$.

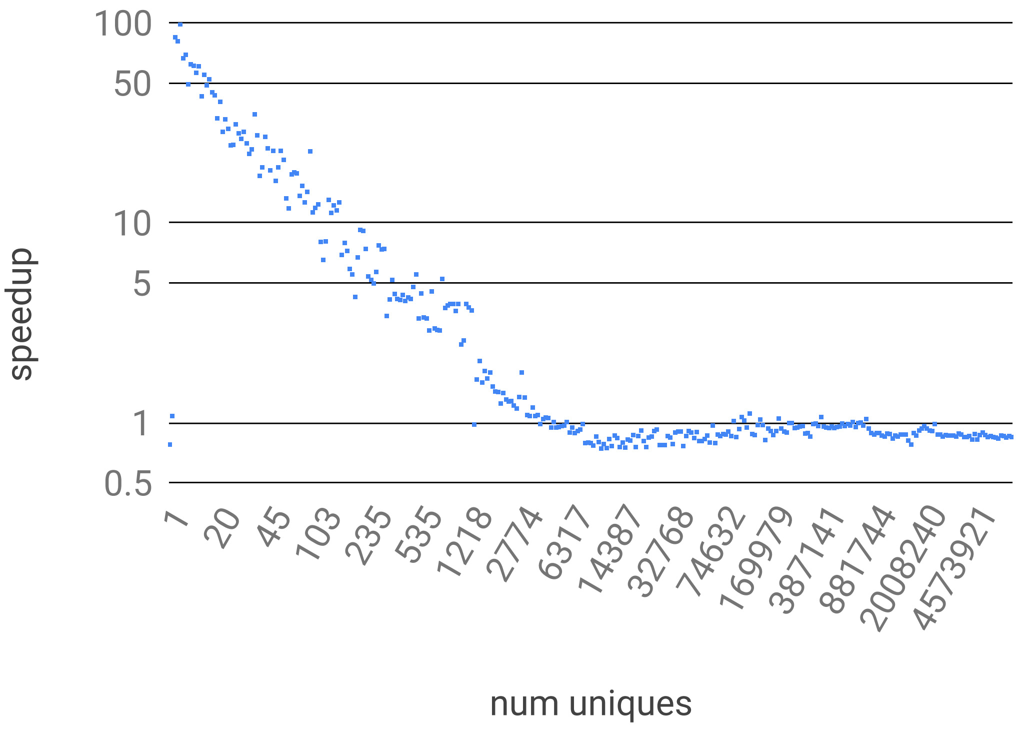


Figure 2.12: Throughput speedup of eager ($\epsilon = 0.04$) vs no-eager ($\epsilon = 1.0$) propagation, $k = 4096$.

Chapter 3

Intermediate Value Linearizability: A Quantitative Correctness Criterion

3.1 Introduction

3.1.1 Motivation

Sketches are quantitative objects that support UPDATE and QUERY operations, where the return value of a QUERY is from an ordered set. They are essentially succinct (sublinear) summaries of a data stream. For example, a sketch might estimate the number of packets originating from any IP address, without storing a record for every possible address. Typical sketches are *probably approximately correct (PAC)*, estimating some aggregate quantity with an error of at most ϵ with probability at least $1 - \delta$ for some parameters ϵ and δ. We say that such sketches are (ϵ, δ)*-bounded*.

The ever increasing rates of incoming data create a strong demand for parallel stream processing [32, 61]. In order to allow queries to return fresh results in real-time without hampering data ingestion, it is paramount to support queries concurrently with updates [99, 111]. But parallelizing sketches raises some important questions, for instance: *What are the semantics of overlapping operations in a concurrent sketch?*, *How can we prove error guarantees for such a sketch?*, and, in particular, *Can we reuse the myriad of clever analyses of existing sketches' error bounds in parallel settings without opening the black box?* In this paper we address these questions.

3.1.2 Our contributions

The most common correctness condition for concurrent objects is linearizability. Roughly speaking, it requires each parallel execution to have a *linearization*, which is a sequential execution of the object that "looks like" the parallel one. (See Section 3.2 for a formal definition.) But sometimes linearizability is too restrictive leading to a high implementation cost, as is shown in this paper and motivates other works on relaxing linearizability [59, 6, 92, 24, 3].

In Section 3.3, we propose *Intermediate Value Linearizability (IVL)*, a new correctness criterion for quantitative objects. Intuitively, the return value of an operation of an IVL object is bounded between two legal values that can be returned in linearizations. The motivation for allowing this is that if the system designer is happy with either of the legal values, then the intermediate value should also be fine. For example, consider a system where processes count events, and a monitoring process detects when the number of events passes a threshold. The monitor constantly reads a shared counter, which other processes increment in batches. If an operation increments the counter from 4 to 7 batching three events, IVL allows a concurrent read by the monitoring process to return 6, although there is no linearization in which the counter holds 6. We formally define IVL and prove that this property is *local*, meaning that a history composed of IVL objects is itself IVL. This allows reasoning about single objects rather than about the system as a whole. We formulate IVL first for deterministic non-randomized objects, and then extend it to capture randomized ones.

Next, we consider (ϵ, δ)-bounded algorithms like data sketches. Existing (sequential) algorithms have sequential error analyses which we wish to leverage for the concurrent case. In Section 3.4 we formally define (ϵ, δ)-bounded objects, including concurrent ones. We then prove a key theorem about IVL, stating that an IVL implementation of a sequential (ϵ, δ)-bounded object is itself (ϵ, δ)-bounded. The importance of this theorem is that it *provides a generic way to leverage the vast literature on sequential (ϵ, δ)-bounded sketches [89, 48, 25, 82, 31, 4] in concurrent implementations.*

In Section 3.5, we present four examples of IVL objects, each showing a different way in which IVL can be used. We first present a wait-free IVL implementation of a batched counter from single-writer-multi-reader (SWMR) registers with $O(1)$ step complexity for UPDATE operations (we will later show that linearizable implementations are inherently more costly). We then showcase an (ϵ, δ)-bounded object, via the example of a concurrent *CountMin (CM)* sketch [31], which estimates the frequencies of items in a data stream. We prove that a straightforward parallelization of this sketch is IVL. By the aforementioned theorem, we deduce that the concurrent sketch adheres to the error guarantees of the original sequential one, without having to "open" the analysis. We note that this parallelization is *not* linearizable. We further show that an r-relaxation of the IVL CM sketch – analogous to r-relaxations of linearizability [59] – allows for efficient concurrent implementations which preserve the sketch's error up to an additive constant.

We then show that data structure iterators returning non-atomic snapshots [12, 86] can be captured via IVL. To this end, we augment their specification and implementation with an *auxiliary history variable*, holding "tombstones" for deleted items. Note that we add the auxiliary variable both at the concrete level (the data structure is augmented to track its removals in an auxiliary variable holding tombstones) and at the abstract level (the iterator in the augmented sequential specification returns a set including tombstones). The algorithm augmented with the auxiliary variable is an IVL implementation of the augmented sequential specification. We show that this specification captures the standard notion of non-atomic iterators [12, 86]. Our last example illustrates that in some cases, IVL needs to be paired with additional correctness criteria, and discuss the example of an IVL and sequentially consistent [105] priority queue.

Finally, we show that IVL is sometimes inherently cheaper than linearizability. We illustrate this in Section 3.6 via the example of a *batched counter*. We prove a lower bound of $\Omega(n)$ step complexity for the UPDATE operation of any wait-free linearizable implementation, using only SWMR registers. As our IVL implementation has a step complexity of $O(1)$ for UPDATE operations, this exemplifies that there is an inherent and unavoidable cost when implementing linearizable algorithms, which can be circumvented by implementing IVL algorithms instead.

3.2 Preliminaries

Section 3.2.1 discusses deterministic shared memory objects and defines linearizability. In Section 3.2.2 we discuss randomized algorithms and their correctness criteria.

3.2.1 Deterministic objects

We consider a standard shared memory model [60], where a set of *asynchronous processes* access atomic shared memory variables. Accessing these shared variables is instantaneous. Processes take *steps* according to an *algorithm*, which is a deterministic state machine, where a step can access a shared memory variable, do local computations, and possibly return some value. A *state* of the system is an assignment of values to all shared and local variables. An *execution* of an algorithm is an alternating sequence of steps and states. We focus on algorithms that implement *objects*, which support *operations*, such as READ and WRITE. Operations begin with an *invocation* step and end with a *response* step. A *schedule*, denoted σ, is the order in which processes take steps, and the operations they invoke in invoke steps with their parameters. Because we consider deterministic algorithms, σ uniquely defines an execution of a given algorithm.

A *history* is the sequence of invoke and response steps in an execution. Given an algorithm A and a schedule σ, $H(A, \sigma)$ is the history of the execution of A with schedule σ. A *sequential* history is an alternating sequence of invocations and their responses, beginning with an invoke step. We denote the return value of operation *op* with parameter *arg* in history H by $\text{ret}(x.op, H)$, where x is the object exposing operation *op*. We refer to the invocation step of operation *op* with parameter *arg* by process p as $inv_p(x.op(arg))$ and to its response step by $rsp_p(x.op(arg) \to ret)$, where

$ret = \mathrm{ret}(x.op, H)$, and where x in the object exposing operation op. We omit x and arg when obvious from context. A history defines a partial order on operations: Operation op_1 *precedes* op_2 in history H, denoted $op_1 \prec_H op_2$, if $rsp(op_1)$ precedes $inv(op_2(arg))$ in H. Two operations are *concurrent* if neither precedes the other.

A *well-formed* history is one that does not contain concurrent operations by the same process, and where every response event for operation op is preceded by an invocation of the same operation. A schedule is well-formed if it gives rise to a well-formed history, and an execution is well-formed if it is based on a well-formed schedule. We denote by $H|_x$ the sub-history of H consisting only of invocations and responses on object x. Operation op is *pending* in a history H if op is invoked in H but does not return.

Correctness of an object's implementation is defined with respect to a sequential specification $\mathcal{H}$, which is the object's set of allowed sequential histories. If the history spans multiple objects, $\mathcal{H}$ consists of sequential histories H such that for all objects x, $H|_x$ pertains to x's sequential specification (denoted $\mathcal{H}_x$). A *linearization* [60] of a concurrent history H is a sequential history H' such that (1) after removing some pending operations from H and completing others, it contains the same invocations and responses as H' with the same parameters and return values, and (2) H' preserves the partial order $\prec_H$. Note that our definition of linearization diverges from the one in [60] in that it is not associated with any sequential specification; instead we require that the linearization pertain to the sequential specification when defining linearizability as follows: Algorithm A is a *linearizable implementation* of a sequential specification $\mathcal{H}$ if every history of a well-formed execution of A has a linearization in $\mathcal{H}$. An object is *total* if every history H containing a pending operation can be extended by a response to a history H', where $H' \in \mathcal{H}$. In this paper we consider only total objects [60].

3.2.2 Randomized algorithms

In randomized algorithms, processes have access to coin flips from some domain Ω. Every execution is associated with a coin flip vector $\vec{c} = (c_1, c_2, \dots)$, where $c_i \in \Omega$ is the i^{th} coin flip in the execution. A *randomized algorithm* A is a probability distribution over deterministic algorithms $\{A(\vec{c})\}_{\vec{c} \in \Omega^\infty}$[1], arising when A is instantiated with different coin flip vectors. We denote by $H(A, \vec{c}, \sigma)$ the history of the execution of randomized algorithm A observing coin flip vector $\vec{c}$ in schedule σ.

Golab et al. show that randomized algorithms that use concurrent objects require a stronger correctness criterion than linearizability, and propose *strong linearizability* [53]. Roughly speaking, strong linearizability stipulates that the mapping of histories to linearizations must be prefix-preserving, so that future coin flips cannot impact the linearization order of earlier events. In Definition 3.3.4, we capture this notion by requiring that the adversary decide on linearization points before observing the coin-

[1]We do not consider non-deterministic objects in this paper.

flips.

3.3 Intermediate value linearizability

Section 3.3.1 introduces definitions that we utilize to define IVL. Section 3.3.2 defines IVL for deterministic algorithms and proves that it is a local property. Section 3.3.3 extends IVL for randomized algorithms, and Section 3.3.5 compares IVL to other correctness criteria.

3.3.1 Definitions

Our definitions use the notion of skeleton histories: A *skeleton history* is a sequence of invocation and response events, where the return values of the responses are undefined, denoted ?. For a history H, we define the operator $H^?$ as altering all response values in H to ?, resulting in a skeleton history.

In this paper we discuss correctness criteria for a class of objects we call *quantitative.* These are objects that support two operations: (1) UPDATE, which captures all mutating operations that do not return a value; and (2) QUERY, which returns a value from some *partially ordered set (poset).* A poset defines a partial relation between elements from the set, denoted $\leq$. In Section 3.5 we exemplify our definitions for three types of return values: (1) numerical values (a totally ordered domain), (2) sets of elements ordered by containment (a partially ordered domain), and (3) ranked items stored in a priority queue (a totally ordered domain).

In a *deterministic quantitative object* the return values of QUERY operations are uniquely defined. Namely, the object's sequential specification $\mathcal{H}$ contains exactly one history for every sequential history skeleton H; we denote this history by $\tau_{\mathcal{H}}(H)$. Thus, $\tau_{\mathcal{H}}(H^?) = H$ for every history $H \in \mathcal{H}$. Furthermore, for every sequential skeleton history H, by definition, $\tau_{\mathcal{H}}(H) \in \mathcal{H}$. Note that if the object is not deterministic, $\tau_{\mathcal{H}}(H^?)$ isn't uniquely defined. We use the following example to show how skeleton histories can be linearized and then converted back to histories:

Example 3.3.1. Consider an execution in which a batched counter (formally defined in Section 3.6) initialized to 0 is incremented by 3 by process p concurrently with a query by process q, which returns 0. Its history is:

$$H = inv_p(inc(3)), inv_q(query), rsp_p(inc), rsp_q(query \rightarrow 0).$$

The skeleton history $H^?$ is:

$$H^? = inv_p(inc(3)), inv_q(query), rsp_p(inc), rsp_q(query \rightarrow ?).$$

A possible linearization of $H^?$ is:

$$H' = inv_p(inc(3)), rsp_p(inc), inv_q(query), rsp_q(query \rightarrow ?).$$

Given the sequential specification $\mathcal{H}$ of a batched counter, we get:

$$\tau_{\mathcal{H}}(H') = inv_p(inc(3)), rsp_p(inc), inv_q(query), rsp_q(query \rightarrow 3).$$

In a different linearization, the query may return 0 instead.

3.3.2 Intermediate value linearizability

We now define intermediate value linearizability.

Definition 3.3.2 (Intermediate value linearizability)**.** A history H of an object is IVL with respect to sequential specification $\mathcal{H}$ if there exist two linearizations H_1, H_2 of $H^?$ such that for every QUERY Q that returns in H,

$$\text{ret}(Q, \tau_{\mathcal{H}}(H_1)) \leq \text{ret}(Q, H) \leq \text{ret}(Q, \tau_{\mathcal{H}}(H_2)).$$

Algorithm A is an *IVL implementation* of a sequential specification $\mathcal{H}$ if every history of a well-formed execution of A is IVL with respect to $\mathcal{H}$.

Note that a linearizable object is trivially IVL, as the skeleton history of the linearization of H plays the roles of both H_1 and H_2. We emphasize that in a sequential execution, an IVL object is not relaxed in any way – it must follow the sequential specification. Similarly, in a well-formed history, operations of the same process never overlap, and so IVL executions satisfy program order.

We now show that IVL is both local and non-blocking(as defined in [60]):

Lemma 3.3.3. *A history H of a well-formed execution of algorithm A over a set of objects $\mathcal{X}$ is IVL if and only if for each object $x \in \mathcal{X}$, $H|_x$ is IVL.*

Proof. Let A be a deterministic algorithm, and let $\mathcal{H}_x$ be the sequential specification of object x, for every $x \in \mathcal{X}$. The "only if" part is immediate.

Denote by H_1^x, H_2^x the linearizations of ${H|_x}^?$ given by the definition of IVL. We first construct a linearization H_1 of $H^?$, i.e., the order $\prec_{H_1}$, as follows: For every pending operation on object x, we either add the corresponding response or remove it based on H_1^x. We then construct a partial order of operations as the union of $\{\prec_{H_1^x}\}_{x\in\mathcal{X}}$ and the realtime order of operations in H. As each $\prec_{H_1^x}$ must adhere to the realtime order of $H|_x$, and therefore H, and the set of operations $\{\prec_{H_1^x}\}_{x\in\mathcal{X}}$ are disjoint, this partial order is well defined. Consider two concurrent operations op_1, op_2 in H. If they do not belong to the same history $H|_x$ for some object x, we order them arbitrarily in $\prec_{H_1}$. We construct linearization H_2 of H^2 by defining the order of operations $\prec_{H_2}$ in a similar fashion, where the added responses or removed pending operations are based on H_2^x instead of H_1^x.

By construction all invocations and responses appearing in $H^?$ appear both in H_1 and in H_2, and H_1 and H_2 preserve the partial order $\prec_{H^?}$. Therefore, H_1 and H_2 are linearizations of $H^?$.

Consider some read R on some object $x \in \mathcal{X}$ that returns in H. As $H|_x$ is IVL $\text{ret}(R, \tau_{\mathcal{H}_x}(H_1^x)) \leq \text{ret}(R, H|_x) \leq \text{ret}(R, \tau_{\mathcal{H}_x}(H_2^x))$. Note that $\text{ret}(R, H|_x) = \text{ret}(R, H)$. Furthermore, $\text{ret}(R, \tau_{\mathcal{H}_x}(H_1^x)) = \text{ret}(R, \tau_{\mathcal{H}}(H_1))$ and $\text{ret}(R, \tau_{\mathcal{H}_x}(H_2^x)) = \text{ret}(R, \tau_{\mathcal{H}}(H_2))$, as objects other than x do not affect the return value of this operation. Therefore H is IVL. $\square$

Locality allows system designers to reason about their system in a modular fashion. Each object can be built separately, and the system as a whole still satisfies the property.

Theorem 3.1. *Let H be an IVL history of a well-formed execution of a algorithm A. If $inv_p(op(arg))$ is an invocation of a pending operation in H, then there exists $rsp_p(op) \to ret$ such that $H \cdot rsp_p(op) \to ret$ is IVL.*

Proof. Let H be an IVL history of a well-formed execution of a algorithm A. Let $inv_p(op(arg))$ be an invocation of a pending operation in H. If the invocation is not that of a QUERY operation, then $H \cdot rsp_p(op) \to \bot$ is IVL. Otherwise, denote the query as Q.

Let $H^?$ be the skeleton history of H. As A implements a total object, H can be extended with a response to Q, and therefore $H^?$ can be extended with $rsp_p(op) \to ?$. Denote $H' = H^? \cdot rsp_p(op) \to ?$. Note that H' is also a skeleton history. Let H'' be some linearization of H', and let v be the value $\text{ret}(Q, \tau_{\mathcal{H}}(H''))$. Therefore, $H \cdot rsp_p(op) \to v$ is IVL, where $H_1 = H_2 = H''$. $\square$

3.3.3 Extending IVL for randomized algorithms

For the remainder of this paper, consider algorithms in which the number of steps taken in an operation is independent of the coin flips. We say that these algorithms have *uniform step complexity.* For example, an operation that returns the value of a coin flip is uniform – the number of steps taken is independent of the coin flips. If coin flips are used for control, the algorithm may require padding shorter execution paths with dummy operations. Note that every sequential algorithm that terminates within a bounded number of steps may be converted to have uniform step complexity. Such an algorithm can then be parallelized using locks or in a lock-free manner; the number of steps in a concurrent execution of the resulting concurrent algorithm might be unbounded under certain schedules, but remains independent of the coin flips. The main point of choosing uniform step complexity is that *coin flips have no impact on operations' execution times.*

In a randomized algorithm A with uniform step complexity, every invocation of a given operation returns after the same number of steps, regardless of the coin flip vector $\vec{c}$. This, in turn, implies that for a given σ, for any $\vec{c}, \vec{c}' \in \Omega^\infty$, the arising histories $H(A, \vec{c}, \sigma)$ and $H(A, \vec{c}', \sigma)$ differ only in the operations' return values but not in the order of invocations and responses, as the latter is determined by σ, so their skeletons are equal. For randomized algorithm A and schedule σ, we denote this arising skeleton history by $H^?(A, \sigma)$.

We are faced with a dilemma when defining the specification of a randomized algorithm A, as the algorithm itself is a distribution over a set of algorithms $\{A(\vec{c})\}_{\vec{c}\in\Omega^\infty}$. Without knowing the observed coin flip vector $\vec{c}$, the execution behaves unpredictably. We therefore define a deterministic sequential specification $\mathcal{H}(\vec{c})$ for each coin flip vector $\vec{c} \in \Omega^\infty$, so the sequential specification is a probability distribution on a set of sequential histories $\{\mathcal{H}(\vec{c})\}_{\vec{c}\in\Omega^\infty}$.

A correctness criterion for randomized objects needs to capture the property that the distribution of a randomized algorithm's outcomes matches the distribution of behaviors allowed by the specification. Consider, e.g., some sequential skeleton history H of an object defined by $\{\mathcal{H}(\vec{c})\}_{\vec{c}\in\Omega^\infty}$. Let Q be a query that returns in H, and assume that Q has some probability p to return a value v in $\tau_{\mathcal{H}(\vec{c})}(H)$ for a randomly sampled $\vec{c}$. Intuitively, we would expect that if a randomized algorithm A "implements" the specification $\{\mathcal{H}(\vec{c})\}_{\vec{c}\in\Omega^\infty}$, then Q has a similar probability to return v in sequential executions of A with the same history, and to some extent also in concurrent executions of A of which H is a linearization. In other words, we would like the distribution of outcomes of A to match the distribution of outcomes in $\{\mathcal{H}(\vec{c})\}_{\vec{c}\in\Omega^\infty}$. For example, consider a sequential specification that allows returns 0 with probability 0.5, and 1 with probability 0.5. For an algorithm to implement such an object, it is not enough for it to return only 0 or 1, but we expect 0 to be returned with probability 0.5, and 1 to be returned with probability 0.5.

We observe that in order to achieve this, it does not suffice to require that each history has an arbitrary linearization as we did for deterministic objects, because this might not preserve the desired distribution as is shown in [53]. Instead, for randomized objects we require a common linearization for each skeleton history that will hold true under all possible coin flip vectors. In other words, the linearization is independent of the coin flip vector. We therefore formally define IVL for randomized objects as follows:

Definition 3.3.4 (IVL for randomized algorithms)**.** Consider a skeleton history $H = H^?(A, \sigma)$ of some randomized algorithm A with schedule σ. *H is IVL* with respect to $\{\mathcal{H}(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ if there exist linearizations H_1, H_2 of H such that for every coin flip vector $\vec{c}$ and query Q that returns in H,

$$\mathrm{ret}(Q, \tau_{\mathcal{H}(\vec{c})}(H_1)) \le \mathrm{ret}(Q, H(A, \vec{c}, \sigma)) \le \mathrm{ret}(Q, \tau_{\mathcal{H}(\vec{c})}(H_2)).$$

Algorithm A is an *IVL implementation* of a sequential specification distribution

$\{\mathcal{H}(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ if every skeleton history of a well-formed execution of A is IVL with respect to $\{\mathcal{H}(\vec{c})\}_{\vec{c}\in\Omega^\infty}$.

Since we require a common linearization of the skeleton histories under all coin flips vectors, the linearizations are a foriori independent of future coin flips. Hence, an adversary cannot affect the linearization based on observing the coin flip vector.

3.3.4 r-relaxed IVL

Henzinger et al. define r-relaxed semantics [59]. Intuitively, each operation on the relaxed object is assigned a *cost.* An r-relaxation allows operations whose costs do not exceed r. Rinberg et al. [99] define the cost of some operation o to be the number of operations that precede it in the concurrent history, but do not precede it in the sequential one. Paraphrasing the definition of the r-relaxed specification for some sequential specification $\mathcal{H}$ from Rinberg et al. [99]:

Definition 3.3.5 (r-relaxation)**.** A sequential history H is an *r-relaxation* of a sequential history H' if H is comprised of all the invocations in H' and their responses, and each invocation in H is preceded by all but at most r of the invocations that precede the same invocation in H'. The *r-relaxation* of $\mathcal{H}$ is the set of histories that have r-relaxations in $\mathcal{H}$:
$\mathcal{H}^r \triangleq \{H' | \exists H \in\mathcal{H} \text{ s.t. } H \text{ is an } r\text{-relaxation of } H'\}$.

We can plug Definition 3.3.5 into Definition 3.3.4 instead of the sequential specification there to get a specification for r-relaxed IVL objects. For completeness, we define r-relaxed IVL:

Definition 3.3.6 (r-Relaxed IVL)**.** Consider a skeleton history $H = H^?(A, \sigma)$ of some randomized algorithm A with schedule σ. H is r-relaxed *IVL* with respect to $\{\mathcal{H}^r(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ if there exist linearizations H_1, H_2 of H such that for every coin flip vector $\vec{c}$ and query Q that returns in H,

$$\text{ret}(Q, \tau_{\mathcal{H}^r(\vec{c})}(H_1)) \le \text{ret}(Q, H(A, \vec{c}, \sigma)) \le \text{ret}(Q, \tau_{\mathcal{H}^r(\vec{c})}(H_2)).$$

Algorithm A is an r-relaxed *IVL implementation* of a sequential specification distribution $\{\mathcal{H}(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ if every skeleton history of a well-formed execution of A is IVL with respect to $\{\mathcal{H}^r(\vec{c})\}_{\vec{c}\in\Omega^\infty}$.

As Definition 3.3.6 is IVL with respect to an r-relaxed specification, it is both local and non-blocking. In Section 3.5.2, we show that this allows us to create more NUMA-friendly objects.

3.3.5 Relationship to other relaxations

In spirit, IVL resembles the *regularity* correctness condition for single-writer registers [78], where a query must return either a value written by a concurrent write or the last value written by a write that completed before the query began. Stylianopoulos et al. [111] adopt a similar condition for data sketches, which they informally describe as follows: "a query takes into account all completed insert operations and possibly a subset of the overlapping ones." If the object's estimated quantity (return value) is monotonically increasing throughout every execution, then IVL essentially captures this condition, while also allowing intermediate steps of a single update to be observed. But this is not the case in general. Consider, for example, an object supporting increment and decrement, and a query that occurs concurrently with an increment and an ensuing decrement. If the query takes only the decrement into account (and not the increment), it returns a value that is smaller than all legal return values that may be returned in linearizations, which violates IVL. Our interval-based formalization is instrumental to ensuring that a concurrent IVL implementation preserves the probabilistic error bounds of the respective sequential sketch, which we prove in the next section.

Another example of an object specified in the spirit of IVL is Lamport's monotonic clock [77], where a read is required to return a value bounded between the clock's values at the beginning and end of the read's interval.

Previous work on set-linearizability [92] and interval-linearizability [24] has also relaxed linearizability, allowing a larger set of return values in the presence of overlapping operations. The set of possible return values, however, must be specified in advance by a given state machine; operations' effects on one another must be predefined. In fact, interval-linearizability could be used to define IVL on a per-object basis, by defining a nondeterministic interval-sequential object in which a read operation can return any value in the interval defined by the update operations that are concurrent with it. In contrast, when considering quantitative objects, IVL is generic and does not require additional object-specific definitions; it provides an intuitive quantitative bound on possible return values.

Henzinger et al. [59] define the quantitative relaxation framework, which allows executions to differ from the sequential specification up to a bounded cost function. We use this framework when defining relaxed IVL. Alistarh et al. expand upon this and define *distributional linearizability* [6], which requires a distribution over the internal states of the object for its error analysis. Rinberg et al. consider strongly linearizable r-relaxed semantics for randomized objects [99]. IVL differs from these definitions in two points: First, a sequential history of an IVL object must adhere to the sequential specification, whereas in these relaxations even a sequential history may diverge from the specification. The second is that these relaxations are measured with respect to a single linearization. We, instead, bound the return value between two legal linearizations. The latter is the key to preserving the error bounds of sequential objects, as we next show.

3.4 (ϵ, δ)-bounded objects

In this section we show that for a large class of randomized objects, IVL concurrent implementations preserve the error bounds of the respective sequential ones. More specifically, we focus on randomized objects like data sketches, which estimate some quantity (or quantities) with probabilistic guarantees. Sketches generally support two operations: UPDATE(a), which processes element a, and QUERY(arg), which returns the quantity estimated by the sketch as a function of the previously processed elements. Sequential sketch algorithms typically have probabilistic error bounds. For example, the Quantiles sketch estimates the rank of a given element in a stream within $\pm\epsilon n$ of the true rank, with probability at least $1 - \delta$ [4], where n is the number of completed UPDATE operations that precede the query.

We consider in this section a general class of (ϵ, δ)-*bounded objects* capturing PAC algorithms. A bounded object's behavior is defined relative to a deterministic sequential specification $\mathcal{I}$, which uniquely defines the *ideal* return value for every query in a sequential execution. In an (ϵ, δ)-bounded $\mathcal{I}$ object, each query returns the ideal return value within an error of at most ϵ with probability at least $1 - \delta$. More specifically, it over-estimates (and similarly under-estimates) the ideal quantity by at most ϵ with probability at least $1 - \frac{\delta}{2}$. Formally:

Definition 3.4.1. A sequential randomized algorithm A implements an (ϵ, δ)-bounded $\mathcal{I}$ object if for every query Q returning in an execution of A with any schedule σ and a randomly sampled coin flip vector $\vec{c} \in \Omega^\infty$,

$$ret(Q, H(A, \sigma, \vec{c})) \geq ret(Q, \tau_{\mathcal{I}}(H^?(A, \sigma)) - \epsilon \text{ with probability at least } 1 - \frac{\delta}{2},$$

and

$$ret(Q, H(A, \sigma, \vec{c})) \leq ret(Q, \tau_{\mathcal{I}}(H^?(A, \sigma)) + \epsilon \text{ with probability at least } 1 - \frac{\delta}{2}.$$

A induces a sequential specification $\{A(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ of an (ϵ, δ)-bounded $\mathcal{I}$ object, by defining the return values of all operations in a given history under a give coin flip vector.

We next discuss concurrent implementations of this specification.

To this end, we must specify a correctness criterion on the object's concurrent executions. As previously stated, the standard notion is (strong) linearizability, stipulating that we can "collapse" each operation in the concurrent schedule to a single point in time. Intuitively, this means that every query returns a value that could have been returned by the randomized algorithm at some point during its execution interval. So the query returns an (ϵ, δ) approximation of the ideal value at that particular point. But this point is arbitrarily chosen, meaning that the query may return an ϵ approximation of any value that the ideal object takes during the query's execution. We therefore look at the minimum and maximum values that the ideal object may take during a query's interval, and bound the error relative to these values.

We first define these minimum and maximum values as follows: For a history H, denote by $\mathcal{L}(H^?)$ the set of linearizations of $H^?$. For a query Q that returns in H and an ideal specification $\mathcal{I}$, we define:

$$v^{\mathcal{I}}_{min}(H,Q) \triangleq \min\{\text{ret}(Q,\tau_{\mathcal{I}}(L) \mid L \in \mathcal{L}(H^?)\}); v^{\mathcal{I}}_{max}(H,Q) \triangleq \max\{\text{ret}(Q,\tau_{\mathcal{I}}(L) \mid L \in \mathcal{L}(H^?))\}.$$

Note that even if H is infinite and has infinitely many linearizations, because Q returns in H, it appears in each linearization by the end of its execution interval, and therefore Q can return a finite number of different values in these linearizations, and so the minimum and maximum are well-defined. Correctness of concurrent (ϵ,δ)-bounded objects is then formally defined as follows:

Definition 3.4.2. A concurrent randomized algorithm A implements an (ϵ,δ)-bounded $\mathcal{I}$ object if for every query Q returning in an execution of A with any schedule σ and a randomly sampled coin flip vector $\vec{c} \in \Omega^\infty$,

$$ret(Q, H(A,\sigma,\vec{c})) \geq v^{\mathcal{I}}_{min}(H(A,\sigma,\vec{c}),Q) - \epsilon \text{ with probability at least } 1-\frac{\delta}{2},$$

and

$$ret(Q, H(A,\sigma,\vec{c})) \leq v^{\mathcal{I}}_{max}(H(A,\sigma,\vec{c}),Q) + \epsilon \text{ with probability at least } 1-\frac{\delta}{2}.$$

For simplicity, the definition above uses a common ϵ for all queries. While in some algorithms, ϵ depends on the stream size, i.e., the number of updates preceding a query, to avoid cumbersome notations we use a single variable ϵ, which should be set to the maximum value that the sketch's ϵ bound takes during the query's execution interval. Since the query returns, its execution interval is necessarily bounded, and so ϵ is bounded.

The following theorem shows that IVL implementations allow us to leverage the "legacy" analysis of a sequential object's error bounds.

Theorem 3.2. *If A implements an (ϵ,δ)-bounded $\mathcal{I}$ object (Definition 3.4.1), and A' is an IVL implementation of A (Definition 3.3.4), then A' implements a concurrent (ϵ,δ)-bounded $\mathcal{I}$ object (Definition 3.4.2). Consider a sequential specification $\{A(\vec{c})\}_{\vec{c}\in\Omega^\infty}$ of an (ϵ,δ)-bounded $\mathcal{I}$ object (Definition 3.4.1). Let A' be an IVL implementation of A (Definition 3.3.4). Then A' implements a concurrent (ϵ,δ)-bounded $\mathcal{I}$ object (Definition 3.4.2).*

Proof. Consider a skeleton history $H = H^?(A',\sigma)$ of A' with some schedule σ, and a query Q that returns in H. As A' is an IVL implementation of A, there exist linearizations H_1 and H_2 of H, such that for every $\vec{c} \in \Omega^\infty$, $\mathrm{ret}(Q, \tau_{A(\vec{c})}(H_1)) \leq \mathrm{ret}(Q, H(A, \sigma, \vec{c})) \leq \mathrm{ret}(Q, \tau_{A(\vec{c})}(H_2))$. As A implements a sequential (ϵ, δ)-bounded $\mathcal{I}$ object, $\mathrm{ret}(Q, \tau_{A(\vec{c})}(H_i)$ is bounded as follows:

$$\mathrm{ret}(Q, \tau_{A(\vec{c})}(H_1)) \geq \mathrm{ret}(Q, \tau_{\mathcal{I}}(H_1)) - \epsilon \text{ with probability at least } 1 - \frac{\delta}{2},$$

and

$$\mathrm{ret}(Q, \tau_{A(\vec{c})}(H_2)) \leq \mathrm{ret}(Q, \tau_{\mathcal{I}}(H_2)) + \epsilon \text{ with probability at least } 1 - \frac{\delta}{2}.$$

Furthermore, by definition of v_{min} and v_{max}:

$$\mathrm{ret}(Q, \tau_{\mathcal{I}}(H_1)) \geq v^{\mathcal{I}}_{min}(H(A', \sigma, \vec{c}), Q); \mathrm{ret}(Q, \tau_{A(\vec{c})}(H_2)) \leq v^{\mathcal{I}}_{max}(H(A', \sigma, \vec{c}), Q).$$

Therefore, with probability at least $1 - \frac{\delta}{2}$, $\mathrm{ret}(Q, H(A', \sigma, \vec{c})) \geq v^{\mathcal{I}}_{min}(H(A', \sigma, \vec{c}), Q) - \epsilon$ and with probability at least $1 - \frac{\delta}{2}$, $\mathrm{ret}(Q, H(A', \sigma, \vec{c})) \leq v^{\mathcal{I}}_{max}(H(A', \sigma, \vec{c}), Q) + \epsilon$, as needed.

$\square$

While easy to prove, Theorem 3.2 shows that IVL is in some sense the "natural" correctness property for (ϵ, δ)-bounded objects; it shows that IVL is a sufficient criterion for keeping the error bounded. It is less restrictive – and as we show below, sometimes cheaper to implement – than linearizability, and yet strong enough to preserve the salient properties of sequential executions of (ϵ, δ)-bounded objects. As noted in Section 3.3.5, previously suggested relaxations do not inherently guarantee that error bounds are preserved. For example, regular-like semantics, where a query "sees" some subset of the concurrent updates [111], satisfy IVL (and hence bound the error) for monotonic objects albeit not for general ones. Indeed, if object values can both increase and decrease, the results returned under such regular-like semantics can arbitrarily diverge from possible sequential ones.

The importance of Theorem 3.2 is that it allows us to leverage the vast literature on sequential (ϵ, δ)-bounded objects [89, 48, 25, 82, 31, 4] in concurrent implementations. We illustrate this in Section 3.3.4 below via the example of an IVL parallelization of a popular data sketch. By Theorem 3.2, it preserves the original sketch's error bounds.

3.5 IVL Examples

In this section we present four examples of IVL objects: a simple – and cheap – batched counter in Section 3.5.1, an r-relaxed (ϵ, δ)-bounded CountMin sketch in Section 3.5.2, a non-atomic iterator in Section 3.5.3, and a priority queue in Section 3.5.4.

3.5.1 Shared batched counter

We now show an example where IVL is inherently less costly than linearizability. We implement an IVL batched counter, and show that its UPDATE operation has step complexity $O(1)$. The algorithm uses single-writer-multi-reader (SWMR) registers, which are registers which only a single process can write to but all the processes can read from. In Section 3.6, we prove that all *linearizable* implementations of a batched counter using SWMR registers incur step complexity $\Omega(n)$ for the UPDATE operation. This is in contrast with standard (non-batched) counters, which can be implemented with a constant update time. Intuitively, the difference is that in a standard counter, all intermediate values “occur” in an execution (provided that return values are all integers and increments all add one), and so all values allowed by IVL are also allowed by linearizability.

A *batched counter* object supports the operations UPDATE(v) where $v \geq 0$, and READ$()$. Note that the counter's value is monotonic, as it supports only positive increments. The sequential specification for this object is simple: a READ operation returns the sum of all values passed to UPDATE operations that precede it, and 0 if no UPDATE operations were invoked. The UPDATE operation returns nothing. When the object is shared, we denote an invocation of UPDATE by process i as UPDATE$_i$. We denote the sequential specification of the batched counter by $\mathcal{H}$.

Algorithm 4 Algorithm for process p_i, implementing an IVL batched counter.

```
1: shared array $v[1 \ldots n]$ initialized to $[0, 0, \ldots, 0]$
2: **procedure** UPDATE$_i(v)$
3:     $v[i] \leftarrow v[i] + v$
4: **procedure** READ
5:     $sum \leftarrow 0$
6:     **for** $i : 1 \leq i \leq n$ **do**
7:         $sum \leftarrow sum + v[i]$
8:     **return** $sum$
```

Algorithm 4 presents an IVL implementation of a batched counter with n processes using an array v of n SWMR registers. The implementation is a trivial parallelization: an UPDATE operation increments the process's local register while a READ scans all

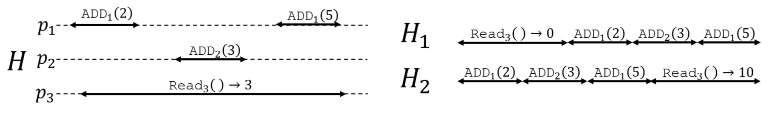


Figure 3.1: A possible concurrent history of the IVL batched counter: p_1 and p_2 update their local registers, while p_3 reads. p_3 returns an intermediate value between the counter's state when it starts, which is 0, and the counter's state when it completes, which is 10. In H_1, the READ operation before all concurrent UPDATE operations, and in H_2 it is ordered after all concurrent ones. Its return value is bounded between those in H_1, where it returns 0, and H_2, where it returns 10.

registers and returns their sum. This implementation is not linearizable because the reader does not take an atomic snapshot of $v[1 \dots n]$. Thus, it may see a later UPDATE and miss an earlier one, as illustrated in Figure 3.1. First, note that the following observation follows from the pseudo-code of Algorithm 4: We now prove the following lemma:

Lemma 3.5.1. *Algorithm 4 is an IVL implementation of a batched counter.*

Proof. Let H be a well-formed history of a schedule σ of Algorithm 4, and let α be its execution. We first complete H be adding appropriate responses to all UPDATE operations that updated v (executed Line 3), and removing all other pending UPDATE and READ operations. We denote this completed history as H'.

Let H_1 be a linearization of $H'^?$ given by ordering UPDATE operations by their return steps, and ordering READ operations after all preceding operations in $H'^?$ and before concurrent UPDATE operations. READ operations assigned to the same point are ordered by their invoke steps. Let H_2 be a linearization of $H'^?$ given by ordering UPDATE operations by their invocations, and ordering READ operations operations after all operations that precede them in $H'^?$ and after concurrent UPDATE operations. READ operations assigned to the same point are ordered by their invoke steps. Let α_i for $i = 1, 2$ be a sequential execution of a batched counter with history $\tau_{\mathcal{H}}(H_i)$.

By construction, H_1 and H_2 are linearizations of $H'^?$ (as they adhere to real-time order). Let R be some READ operation that completes in H. Let $v_\alpha[1 \dots n]$ be the array as read by R in α, $v_{\alpha_1}[1 \dots n]$ as read by R in α_1 and $v_{\alpha_2}[1 \dots n]$ as read by R in α_2. To show that $\text{ret}(R, \tau_{\mathcal{H}}(H_1)) \leq \text{ret}(R, H) \leq \text{ret}(R, \tau_{\mathcal{H}}(H_2))$, we show that $v_1[j] \leq v[j] \leq v_2[j]$ for every index $1 \leq j \leq n$.

For any index j, $v[j]$ is incremented with non-negative values. By the construction of H_1, all UPDATE operations that update $v[j]$ that precede R in H also precede it in H_1. Therefore, $v_{\alpha_1}[j] \leq v_\alpha[j]$. By the construction of H_2, all UPDATE operations that update $v[j]$ that precede R in H also precede it in H_2. Furthermore, in H_2 all concurrent operations that have updated $v[j]$ also precede R. As these are all concurrent UPDATE operations that update $v[j]$, $v_\alpha[j] \leq v_{\alpha_2}[j]$.

As all entries in the array are non-negative, it follows that $\sum_{j=1}^{n} v_{\alpha_1}[j] \leq \sum_{j=1}^{n} v_\alpha[j] \leq \sum_{j=1}^{n} v_{\alpha_2}[j]$, and therefore $\text{ret}(R, \tau_{\mathcal{H}}(H_1)) \leq \text{ret}(R, H) \leq \text{ret}(R, \tau_{\mathcal{H}}(H_2))$. □

This algorithm can efficiently implement a distributed or NUMA-friendly counter, as processes only access their local registers for updaing values thereby lowering the cost of incrementing the counter.Such an implementation is highly efficient when update operations are much more frequent than read operations. This is of great importance, as memory latencies are often the main bottleneck in shared object emulations [84]. As there are no waits in either UPDATE or READ, it follows that the algorithm is wait-free. Furthermore, the READ step complexity is $O(n)$, and the UPDATE step complexity is $O(1)$. Thus, we have shown the following theorem:

Theorem 3.3. *There exists a bounded wait-free IVL implementation of a batched counter using only SWMR registers, such that the step complexity of* UPDATE *is $O(1)$ and the step complexity of* READ *is $O(n)$.*

3.5.2 Relaxed shared CountMin sketch

In this Section we present a relaxed CountMin sketch. Section 3.5.2 presents a concurrent IVL implementation, and shows that this implementation isn't linearizable. Section 3.5.2 presents the r-relaxed IVL CM sketch.

IVL CountMin sketch

Cormode et al. propose the *CountMin (CM)* sketch [31], which estimates the frequency of an item a, denoted f_a, in a data stream, where the data stream is over some alphabet Σ. The CM sketch supports two operations: UPDATE(a), which updates the object based on $a \in \Sigma$, and QUERY(a), which returns an estimate on the number of UPDATE(a) calls that preceded the query. The number of UPDATE operations that precede a query is called the *stream length*, generally denoted by N.

The sequential algorithm's underlying data structure is a matrix c of $w \times d$ counters, for some parameters w, d determined according to the desired error and probability bounds. The sketch uses d hash functions $h_i : \Sigma \mapsto [1, w]$, for $1 \leq i \leq d$. The hash functions are generated using the random coin flip vector $\vec{c}$, and have certain mathematical properties whose details are not essential for understanding this paper. The algorithm's input (i.e., the schedule) is generated by a so-called *weak adversary*, namely, the input is independent of the randomly drawn hash functions.

The CountMin sketch, denoted $CM(\vec{c})$, is illustrated in Figure 3.2, and its pseudocode is given in Algorithm 5. On UPDATE(a), the sketch increments counters $c[i][h_i(a)]$ for every $1 \leq i \leq d$. QUERY(a) returns $\hat{f}_a = \min_{1 \leq i \leq d}\{c[i][h_i(a)]\}$.

Algorithm 5 CountMin($\vec{c}$) sketch.

```
1: array $c[1 \dots d][1 \dots w]$                         ▷ Initialized to 0
2: hash functions $h_1, \dots h_d$        ▷ $h_i : \Sigma \mapsto [1, w]$, initialized using $\vec{c}$

3: procedure UPDATE($a$)
4:     for $i : 1 \leq i \leq d$ do
5:         atomically increment $c[i][h_i(a)]$
6: procedure QUERY($a$)
7:     $min \leftarrow \infty$
8:     for $i : 1 \leq i \leq d$ do
9:         $c \leftarrow c[i][h_i(a)]$
10:        if $min > c$ then $min \leftarrow c$
11:    return $min$
```

Cormode et al. show that, for desired bounds δ and α, given appropriate values of w and d, with probability at least $1 - \delta$, the estimate of a query returning $\hat{f}_a$ is bounded

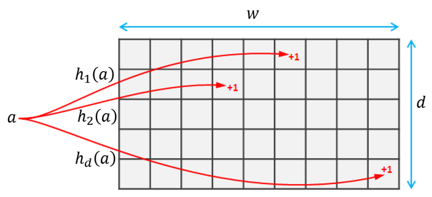


Figure 3.2: An example CountMin sketch, of size $w \times d$, where $h_1(a) = 6$, $h_2(a) = 4$ and $h_d(a) = w$.[2]

by $f_a \leq \hat{f}_a \leq f_a + \alpha N$, where N is the number of elements in the stream and f_a is the ideal value. Thus, for $\epsilon = \alpha N$, CM is a sequential (ϵ, δ)-bounded object. Its sequential specification distribution is $\{CM(\vec{c})\}_{\vec{c} \in \Omega^\infty}$.

Proving an error bound for an efficient parallel implementation of the CM sketch is not trivial. Any linearizable implementation, and even an r-relaxed linearizable, such as that of Rinberg et al. [99], requires the query to take an atomic snapshot of the matrix [94], which we forgo in Algorithm 5. Distributional linearizability [6] necessitates an analysis of the error bounds directly in the concurrent setting, without leveraging the sketch's existing analysis for the sequential setting.

Instead, we utilize IVL to leverage the sequential analysis for a parallelization that is not strongly linearizable (or indeed linearizable), without using a snapshot. Consider the straightforward parallelization of the CM sketch, whereby the operations of Algorithm 5 may be invoked concurrently and each counter is atomically incremented (e.g., using a FAA atomic operation [68]) on line 5 and read on line 9. We call this parallelization PCM. We next prove that it is IVL.

Lemma 3.5.2. *PCM is an IVL implementation of CM.*

Proof. Let H be a history of an execution σ of PCM. Let H_1 be a linearization of $H^?$ such that every query is linearized prior to every concurrent update, and let H_2 be a linearization of $H^?$ such that every query is linearized after every concurrent update. Let σ_i for $i = 1, 2$ be a sequential execution of CM with history H_i. Consider some $Q = $ QUERY(a) that returns in H, and let $U_1, \ldots, U_k$ be the concurrent updates to Q.

Denote by $c_\sigma(Q)[i]$ the value read by Q from $c[i][h_i(a)]$ in line 9 of Algorithm 5 in an execution σ. As processes only increment counters, for every $1 \leq i \leq d$, $c_\sigma(Q)[i]$ is at least $c_{\sigma_1}(Q)[i]$ (the value when the query starts) and at most $c_{\sigma_2}(Q)[i]$ (the value when all updates concurrent to the query complete). Therefore, $c_{\sigma_1}(Q)[i] \leq c_\sigma(Q)[i] \leq c_{\sigma_2}(Q)[i]$.

Consider a randomly sampled coin flip vector $\vec{c} \in \Omega^\infty$. Let j be the loop index the last time query Q alters the value of its local variable *min* (line 10), i.e., the index of the minimum read value. As a query in a history of $CM(\vec{c})$ returns the minimum

[2]Source: `https://stackoverflow.com/questions/6811351/explaining-the-count-sketch-algorithm`, with alterations.

value in the array, $\mathrm{ret}(Q, \tau_{CM(\vec{c})}(H_1)) \le c_{\sigma_1}(Q)[j]$. Furthermore, $\mathrm{ret}(Q, \tau_{CM(\vec{c})}(H_2))$ is at least $c_\sigma(Q)[j]$, otherwise Q would have read this value and returned it instead. Therefore:

$$\mathrm{ret}(Q, \tau_{CM(\vec{c}))}(H_1)) \le \mathrm{ret}(Q, H(PCM, \sigma, \vec{c})) \le \mathrm{ret}(Q, \tau_{CM(\vec{c})}(H_2))$$

As needed. □

Combining Lemma 3.5.2 and Theorem 3.2, and by utilizing the sequential error analysis from [31], we have shown the following corollary:

Corollary 3.4. *Consider a concurrent history H of PCM with parameters (ϵ, δ), with a stream of length N. Let $\hat{f}_a$ be a return value from query Q with parameter a in H. Let f_a^{start} be the ideal frequency of element a at the invocation of Q, and let f_a^{end} be the ideal frequency of element a at the response of Q. Then:*

$$f_a^{start} \le \hat{f}_a \le f_a^{end} + \epsilon \textit{ with probability at least } 1 - \delta.$$

Proof. Let CM be a sequential (ϵ, δ)-bounded object. Lemma 3.5.2 proves that PCM is an IVL implementation of CM, therefore, by Theorem 3.2, PCM implements a concurrent (ϵ, δ)-bounded object.

Consider some concurrent history H containing N update operations, and consider some query Q of element a that returns in H. Let H^{start} be the prefix of H up to the invocation of Q, where all pending operations are removed. Let f_a^{start} be the number of update operations in H^{start} with parameter a. As each update operation increments the counters Q reads, and they all precede Q, then $f_a^{\text{start}} \le v_{min}^{\mathcal{I}}(H, Q)$.

Let H^{end} be the prefix of H up to the response of Q, where all pending operations are completed.Let f_a^{end} be the number of update operations in H^{end} with parameter a. As each update operation increments the counters Q reads, then $f_a^{\text{end}} \ge v_{min}^{\mathcal{I}}(H, Q)$. Therefore:

$$f_a^{\text{start}} - \epsilon \le \hat{f}_a \le f_a^{\text{end}} + \epsilon \text{ with probability at least } 1 - \delta.$$

This can be further improved by noting that the Q returns at least f_a^{start}, as it cannot read the counters with values less than f_a^{start}. Therefore:

$$f_a^{\text{start}} \le \hat{f}_a \le f_a^{\text{end}} + \epsilon \text{ with probability at least } 1 - \delta.$$

□

The following example demonstrates that PCM is not a linearizable implementation of CM.

Example 3.5.3. Consider the following execution σ of PCM: Assume that $\vec{c}$ is such that $h_1(a) = h_2(a) = 1$, $h_1(b) = 2$ and $h_2(b) = 1$. Assume that initially

$$c = \begin{array}{|c|c|}\hline 1 & 4 \\ \hline 2 & 3 \\ \hline \end{array}.$$

First, process p invokes $U = \text{UPDATE}(a)$ which increments $c[1][1]$ to 2 and stalls. Then, process q invokes $Q_1 = \text{QUERY}(a)$ which reads $c[1][1]$ and $c[2][1]$ and returns 2, followed by $Q_2 = \text{QUERY}(b)$ which reads $c[1][2]$ and $c[2][1]$ and returns 2. Finally, process p increments $c[2][1]$ to be 3.

Assume by contradiction that H is a linearization of σ, and $H \in CM(\vec{c})$. The return values imply that $U \prec_H Q_1$ and $Q_2 \prec_H U$. As H is a linearization, it maintains the partial order of operations in σ, therefore $Q_1 \prec_H Q_2$. A contradiction.

Relaxed IVL CM sketch

Using the r-relaxed IVL form, we can create a buffered CM sketch by using a similar framework to Rinberg et al. [99]. The UPDATE operations are buffered locally by updating threads until a certain threshold is reached. The resulting matrix is then merged into the shared CM sketch, which is updated element by element.

This buffered approach allows for far better memory locality. Rather than every UPDATE operating on shared memory, most operations are local. This leads to better cache utilization, and forgoes expensive NUMA memory accesses.

In Rinberg et al., a QUERY takes a strongly linearizable snapshot of the current global state, and applies the query to it. In a CM sketch, a linearizable snapshot is an expensive operation – it requires an atomic snapshot. Using an IVL query forgoes the need for a strongly linearizable snapshot, while retaining error bounds.

The buffered IVL CM sketch is r-relaxed IVL with respect to $\mathcal{H}_{CM}$, where $r = 2Wb$, where W is the number of worker threads and b is the local buffer size (i.e., the number of updates processed between propagations).

The error analysis follows from the relaxation and the definition of IVL. Consider some query Q on a that returns v, and let S^{start} and S^{end} be the state of the system at the start and end of Q's execution, respectively. Let v^{start} be the number of times a appears in the stream before S^{start}, let v^{end} be the number of times a appears in the stream before S^{end}, and let N^{end} be the length of the stream at S^{end}.

$$v^{start} - r \leq v \leq v^{end} + \epsilon N^{end}.$$

3.5.3 Non-atomic iterators

While until now we focused on numerical objects, the idea behind IVL can be expanded to include other types of objects as well. For example, iterators in map and set data structures (like skiplists and search trees) typically return a non-atomic scan of the set of keys – or items – in the data structure [12, 86]. We show that the semantics of such scan operations are naturally captured using IVL. Consider a data structure supporting three operations, (1) INSERT, (2) DELETE, and (3) SCAN. The sequential specification $\mathcal{H}_{MAP}$ is straightforward, a SCAN returns all elements that were inserted before it and were not deleted before it.

The concurrent semantics are typically defined as follows [12, 86]:

Definition 3.5.4 (Non-atomic SCAN operation semantics)**.** Consider a scan operation, returning some set S. Let K be the elements that have been inserted prior to the scan's invocation and have not been deleted before the scan. Let I be the elements being inserted concurrently to the scan, and let D be the set of elements that are removed concurrently to the scan. Then $K \setminus D \subseteq S \subseteq K \cup I$.

This definition resembles IVL with the partial order of set containment. However, IVL also adheres to program order. So if a thread adds element a, then removes it, then adds element b, an IVL scan result cannot contain both a and b, which is allowed by Definition 3.5.4.

We can, nevertheless, use IVL to capture the correctness semantics of such non-atomic iterators as specified by Definition 3.5.4. To this end, we add an auxiliary history variable both at the concrete level (the data structure is augmented to track its removals in an auxiliary history variable holding tombstones) and at the abstract level (the SCAN in the augmented sequential specification returns a set including tombstones). The algorithm augmented with the auxiliary variable is an IVL implementation of the augmented sequential specification. The INSERT(a) operation inserts a into the auxiliary variable. The DELETE(a) operation inserts a tombstone for a, denoted $-a$, into the auxiliary variable. The concrete return value is defined via a function f that returns the set of elements that are included and do not have tombstones in the auxiliary variable, e.g., $f(\{a, -a, b\}) = \{b\}$.

Formally, let $\mathcal{V}$ be the set of all possible elements, and denote the possible tombstones as $\mathcal{V}^- = \{-v | v \in \mathcal{V}\}$. The auxiliary variable is some $S \in 2^{\mathcal{V} \cup \mathcal{V}^-}$. We define $f : 2^{\mathcal{V} \cup \mathcal{V}^-} \mapsto 2^{\mathcal{V}}$ as: $f(S) = \{v | v \in S \wedge -v \notin S\}$.

Given the sequential specification $\mathcal{H}_{MAP}$ defined above, denote $\mathcal{H}_{T-MAP}$ as the augmented object. The sequential specification of a SCAN operation is also straightforward: a SCAN returns a set in $2^{\mathcal{V} \cup \mathcal{V}^-}$ consisting of all elements that were inserted before it, and tombstones for all elements deleted before it.

We prove that IVL semantics with the auxiliary variable is equivalent to Definition 3.5.4:

Lemma 3.5.5. *Consider a history H of a concurrent object implementing $\mathcal{H}_{MAP}$ and some scan of it returning a set of items $S \in 2^{\mathcal{V}}$. Denote by $\mathcal{H}_{T-MAP}$ the object augmented with the auxiliary history variable tracking tombstones. Then $\exists S' \in 2^{\mathcal{V} \cup \mathcal{V}^-}$ such that returning H is IVL with respect to $\mathcal{H}_{T-MAP}$ and $f(S') = S$.*

Proof. Let H be a history of some concurrent object implementing $\mathcal{H}_{MAP}$, and let S

be the result of some non-atomic scan operation in H. Let K,I, and D be as defined by Definition 3.5.4.

S satisfies Definition 3.5.4, as the object implements $\mathcal{H}_{MAP}$. We show that $\exists S'$ such that S' is IVL with respect to $\mathcal{H}_{T-MAP}$ and $f(S') = S$.

Let H_1 be a linearization of $H^?$ where the scan is linearized before all concurrent operations, and let H_2 be a linearization of $H^?$ where the scan is linearized after all concurrent operations. Let ${S'}_i \in 2^{\mathcal{V} \cup \mathcal{V}^-}$ be the return value of the scan in $\tau_{\mathcal{H}}(H_i)$, for $i = 1, 2$. Let K' be the set of all insertions and deletions before the scan in H_1, then, by construction, $S'_1 = K'$ and $S'_2 = K' \cup I \cup D^-$. Note that $f(K') = K$.

We construct S' as $S'_1 \cup I' \cup D'^-$ as follows: $I' = S \cap I$, and $D' = D \setminus S$. I.e., I' is all inserted elements that are observed by S, and D' are all deleted elements not observed by S – meaning elements that are deleted concurrently to the scan and not observed by it. By construction

$$S'_1 \subseteq S' = S'_1 \cup I' \cup D'^- \subseteq S'_1 \cup I \cup D^- = S'_2,$$

so returning S' is IVL with respect to $\mathcal{H}_{T-MAP}$.

Furthermore, by construction, $S' = K' \cup (S \cap I) \cup (D \setminus S)^-$, so $f(S') = K \cup (S \cap I) \setminus (D \setminus S)$. By Definition 3.5.4 $K \setminus D \subseteq S \subseteq K \cup I$, we get that $K \setminus (D \setminus S) \subseteq S \subseteq K \cup (I \cap S)$, and $f(S') = S$, as needed. □

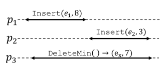
We believe that instead of giving a long definition of the SCAN operation as is currently done [12, 86], IVL captures the semantics of such operations accurately and succinctly.

3.5.4 Sequentially consistent IVL priority queue

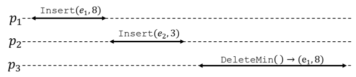
We next show that for more complex (non-numeric) objects, IVL can be used in conjuction with additional properties. Consider the example of a *Priority Queue (PQ)*, which supports two operations, INSERT and DELETEMIN [42, 101]. The sequential PQ holds a set of priority-element pairs. An INSERT(e, p) adds element e with priority p to the set, and a DELETEMIN returns and removes the element with the lowest priority from the set.

The set of pairs is partially ordered by priority, with ties broken by the elements themselves. Thus, we can use IVL to define the PQ's concurrent semantics. However, IVL alone allows the PQ to return elements that were never inserted, as shown in Figure 3.3a. In the example, thread p_1 inserts $(e_1, 8)$ to the PQ and thread p_2 inserts $(e_2, 3)$. Concurrently to the insertions, thread q removes an item from the PQ. IVL allows q to return $(e_x, 6)$, as $(e_2, 3) \leq (e_x, 6) \leq (e_2, 8)$, even though $(e_x, 6)$ may be an element that was never inserted to the PQ. Thus, an IVL PQ is meaningless. Fortunately, we can address this by requiring an additional property along with IVL.

To disallow such spurious elements, we require that the PQ also be *sequentially* *consistent (SC)* [105]. We note that SC and IVL are incomparable: IVL requires that sequential executions adhere to the sequential specification, whereas SC only requires program order. For example, Figure 3.3b shows an SC PQ that isn't IVL, and, as noted above, Figure 3.3a shows an IVL PQ that isn't SC.



(a) IVL PQ that isn't SC – $(e_x, 7)$ is returned even though it was never inserted.



(b) SC PQ that isn't IVL – IVL adheres to real-time order, therefore $(e_2, 3)$ must be returned.

Figure 3.3: Possible histories of a priority queue under IVL only (3.3a) and SC only (3.3b).

Combining the IVL and SC properties yields a PQ that must return elements previously inserted to the PQ and not yet deleted (by SC), yet the DELETEMIN operation may return elements disallowed under linearizability. For example, Figure 3.4 presents a history of an IVL and SC PQ. In the example, thread p_1 inserts $(e_1, 8)$ to the PQ, p_2 then inserts $(e_2, 3)$, and then p_1 inserts $(e_3, 5)$. Concurrently to the insertions, p_3 removes an element from the PQ. The SC property requires that the element be $(e_1, 8)$, $(e_2, 3)$ or $(e_3, 5)$, and the IVL property requires that the element have priority at most 8, and at least 3. For example, the element $(e_3, 5)$ can be returned, which is disallowed under linearizability.

An IVL priority queue is useful when the necessary guarantee is that the popped element is one of the top elements, but not necessarily the top one, e.g., parallel graph processing [54], or belief propagation [5].

3.6 Lower bound for linearizable batched counter

The incentive for using an IVL batched counter instead of a linearizable one stems from a lower bound on the step-complexity of a wait-free linearizable batched counter implementation from SWMR registers. To show the lower bound we first define the binary snapshot object. A *snapshot object* has n components written by separate processes, and allows a reader to capture the shared variable states of all n processes instantaneously. We consider the *binary snapshot object*, in which each state component may be either 0 or 1 [63]. The object supports the UPDATE$_i(v)$ and SCAN operations, where

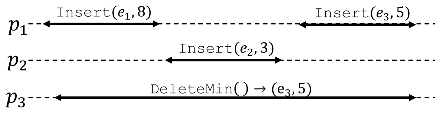


Figure 3.4: A possible concurrent history of a PQ that is both IVL and SC but not linearizable. The SC property requires that the remove be of some previously inserted element, and the IVL property requires that the element have a priority bound between 8 and 3.

the former sets the state of component i to value $v \in \{0, 1\}$ and the latter returns all processes states instantaneously. It is trivial that the SCAN operation must read all states, therefore its lower bound step complexity is $\Omega(n)$. Israeli and Shriazi [70] show that the UPDATE step complexity of any implementation of a snapshot object from SWMR registers is also $\Omega(n)$. This lower bound was shown to hold also for multi writer registers [16]. While their proof was originally given for a multi value snapshot object, it holds in the binary case as well [63].

Algorithm 6 Algorithm for process p_i, solving binary snapshot with a batched counter object.

```
1: local variable $v_i$                                    ▷ Initialized to 0
2: shared batched counter object $BC$                      ▷ Initialized to 0

3: procedure UPDATE$_i(v)$
4:     if $v_i = v$ then return
5:     $v_i \leftarrow v$
6:     if $v = 1$ then $BC$.UPDATE$_i(2^i)$
7:     if $v = 0$ then $BC$.UPDATE$_i(2^n - 2^i)$
8: procedure SCAN
9:     $sum \leftarrow BC$.READ()
10:    $v[0 \ldots n-1] \leftarrow [0 \ldots 0]$           ▷ Initialize an array of 0's
11:    for $i : 0 \leq i \leq n-1$ do
12:        if bit $i$ is set in $sum$ then $v[i] \leftarrow 1$
13:    return $v[0 \ldots n-1]$
```

To show a lower bound on the UPDATE operation of wait-free linearizable batched counters, we show a reduction from a binary snapshot to a batched counter in Algorithm 6. It uses a local variable v_i and a shared batched counter object. In a nutshell, the idea is to encode the value of the i^{th} component of the binary snapshot using the i^{th} least significant bit of the counter. When the component changes from 0 to 1, UPDATE$_i$ adds 2^i, and when it changes from 1 to 0, UPDATE$_i$ adds $2^n - 2^i$. We now prove the following invariant:

Invariant 4. *For any prefix H' of H of length t of a sequential execution of Algorithm 6, the sum held by the counter is $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i$, such that v_i is the parameter passed to the last invocation of* UPDATE$_i$ *in H' if such invocation exists, and* 0 *otherwise, for some integer $c \in \mathbb{N}$.*

Proof. We prove the invariant by induction on the length of H, i.e., the number of invocations in H, denoted t. As H is a sequential history, each invocation is followed by a response. The base if for $t = 0$, i.e., H is the empty execution. In this case no updates have been invoked, therefore $v_i = 0$ for all $0 \leq i \leq n-1$. The sum returned by the counter is 0. Choosing $c = 0$ satisfies the invariant. Our induction hypothesis is that the invariant holds for a history of length t. We prove that it holds for a history of length $t + 1$. The last invocation can be either a SCAN, or an UPDATE(v) by some process p_i. If it is a SCAN, then the counter value doesn't change and the invariant holds. Otherwise, it is an UPDATE(v). Here, we note two cases. Let v_i be p_i's value prior to the UPDATE(v) invocation. If $v = v_i$, then the UPDATE returns without altering the sum and the invariant holds. Otherwise, $v \neq v_i$. We analyze two cases, $v = 1$ and $v = 0$. If $v = 1$, then $v_i = 0$. The sum after the update is $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i + 2^i = c \cdot 2^n + \sum_{i=0}^{n-1} v'_i 2^i$, where $v'_j = v_j$ if $j \neq i$, and $v'_i = 1$, and the invariant holds. If $v = 0$, then $v_i = 1$. The sum after the update is $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i + 2^n - 2^i = (c+1) \cdot 2^n + \sum_{i=0}^{n-1} v'_i 2^i$, where $v'_j = v_j$ if $j \neq i$, and $v'_i = 1$, and the invariant holds. □

Using the invariant, we prove the following lemma:

Lemma 3.6.1. *For any sequential history H, if a* SCAN *returns v_i, and* UPDATE$_i$*(v) is the last update invocation in H prior to the* SCAN*, then $v_i = v$. If no such update exists, then $v_i = 0$.*

Proof. Let S be a SCAN in H'. Consider the sum *sum* as read by scan S. From Invariant 4, the value held by the counter is $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i$. There are two cases, either there is an update invocation prior to S, or there isn't. If there isn't, then by Invariant 4 the corresponding $v_i = 0$. The process sees bit $i = 0$, and will return 0. Therefore, the lemma holds.

Otherwise, there is a an update prior to S in H. As the sum is equal to $c \cdot 2^n + \sum_{i=0}^{n-1} v_i 2^i$, by Invariant 4, bit i is equal to 1 iff the parameter passed to the last invocation of update was 1. Therefore, the scan returns the parameter of the last update and the lemma holds. □

Lemma 3.6.2. *Algorithm 6 implements a linearizable binary snapshot using a linearizable batched counter.*

Proof. Let H be a history of Algorithm 6, and let H' be the subset of operations in H that access the linearizable batched counter (and removed otherwise), where each operation is linearized at its access to the linearizable batched counter with responses added to pending operations, or its response if $v_i = v$ on line 4. Applying Lemma 3.6.1 to H', we get $H' \in \mathcal{H}$ and therefore H is linearizable. □

It follows from the algorithm that if the counter object is bounded wait-free then the SCAN and UPDATE operations are bounded wait-free. Therefore, the lower bound proved by Israeli and Shriazi [70] holds, and the UPDATE must take $\Omega(n)$ steps. Other than the access to the counter in the UPDATE operation, it takes $O(1)$ steps. Therefore, the access to the counter object must take $\Omega(n)$ steps. We have proven the following theorem.

Theorem 3.5. *For any linearizable wait-free implementation of a batched counter object with n processes from SWMR registers, the step-complexity of the* UPDATE *operation is $\Omega(n)$.*

3.7 Conclusion

We have presented IVL, a new correctness criterion that provides flexibility in the return values of quantitative objects while bounding the error that this may introduce. IVL has a number of desirable properties: First, like linearizability, it is a local property, allowing designers to reason about each part of the system separately. Second, also like linearizability but unlike other relaxations of it, IVL preserves the error bounds of PAC objects. Third, IVL is generically defined for all quantitative objects, and does not necessitate object-specific definitions. Finally, IVL is inherently amenable to cheaper implementations than linearizability in some cases.

Via the example of a CountMin sketch, we have illustrated that IVL provides a generic way to efficiently parallelize data sketches while leveraging their sequential error analysis to bound the error in the concurrent implementation.

We have shown that IVL can also capture the semantics of non-atomic snapshots, by augmenting the data structure with an auxiliary history variable. Finally, we have shown that sometimes IVL is useful in tandem with other correctness criteria via the example of a priority queue, where we pair IVL with sequential consistency.

The notion of IVL raises a main question for future research: In this work we have shown that IVL is a sufficient condition for parallel (ϵ, δ)-bounded objects, in that it preserves their sequential error. It would be interesting to investigate whether IVL is also necessary, or whether some weaker condition is sufficient.

Chapter 4

Quancurrent

One of the performance bottlenecks in the generic solution presented in Chapter 2 is the merge operation. Every propagation in the Quantiles sketch requires a sort operation. Furthermore, as the stream size grows, the merger thread may be required to execute multiple sort operations as part of a single propagation. Therefore, as there is a single merger thread, long merges leave update threads idle and harm speedup. We now briefly present Quancurrent, a concurrent quantiles sketch, which is based upon joint work. We go into more detail in our full paper [41]. We first give a brief overview on the importance of the Quantiles sketch.

Understanding the data distribution is a fundamental task in data management and analysis, used in applications such as exploratory data analysis [116], operations monitoring [1], and more. The Quantiles sketch family captures this task [85, 4, 51, 30]. The sketch represents the quantiles distribution in a stream of elements, such that for any $0 \leq \phi \leq 1$, a query for quantile ϕ returns an estimate of the $\lfloor n\phi \rfloor^{\text{th}}$ largest element in a stream of size n. For example, quantile $\phi = 0.5$ is the median. Due to the importance of quantiles approximation, Quantiles sketches are a part of many analytics platforms, e.g., Druid [38], Hillview [23], Presto [95], and Spark [110].

The classic literature on sketches has focused on inducing a small error while using a small memory footprint, in the context of sequential processing: the sketch is built by a single thread, and queries are served only after sketch construction is complete. Only recently, we begin to see works leveraging parallel architectures to achieve a higher ingestion throughput while also enabling queries concurrently with updates [111]. Of these, the only solution suitable for quantiles that we are aware of is the Fast Concurrent Data Sketches (FCDS) framework proposed in Chapter 2. However, the scalability of FCDS-based quantiles sketches is limited unless query freshness is heavily compromised (as we show below). Our goal is to provide a scalable concurrent Quantiles sketch that retains a small error bound with reasonable query freshness.

Quancurrent is based on the solution proposed by Agarwal et. al [4] which is used by Apache DataSketches [11]. Like FCDS, Quancurrent relies on local buffering of stream elements, which are then propagated in bulk to a shared sketch. But Quancurrent improves on FCDS by eliminating the latter's sequential propagation bottleneck, which

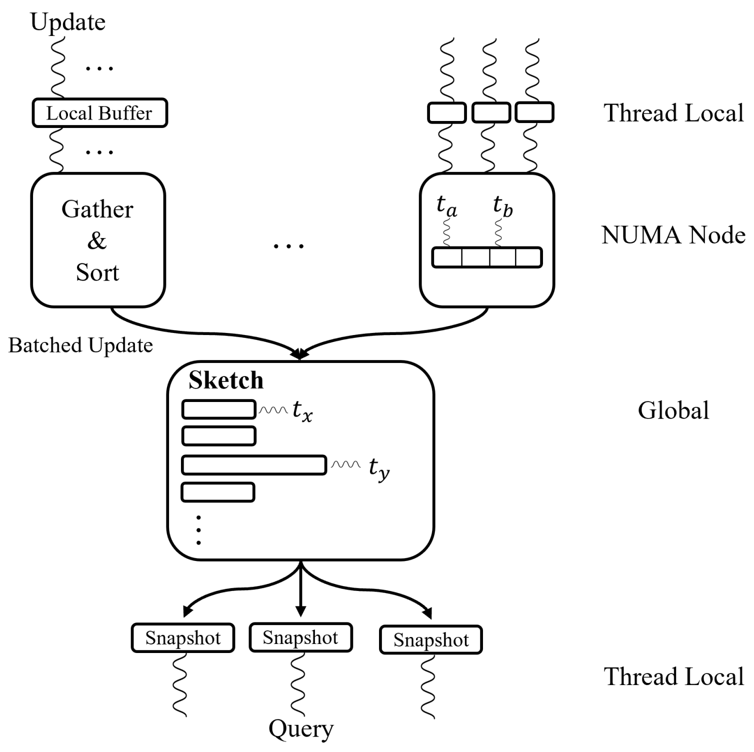


Figure 4.1: Quancurrent's data structures. Threads t_a and t_b are in the same NUMA node and are accessing the *Gather&Sort*, and threads t_x and t_y are executing different propagations.

mostly stems from the need to sort large buffers.

In Quancurrent, sorting occurs at three levels – a small thread-local buffer, an intermediate NUMA-node-local buffer called *Gather&Sort*, and the shared sketch. Moreover, the shared sketch itself is organized in multiple levels, which may be propagated (and sorted) concurrently by multiple threads.

To allow queries to scale as well, Quancurrent serves them from a cached snapshot of the shared sketch. This architecture is illustrated in Figure 4.1. The query freshness depends on the sizes of local and NUMA-local buffers as well as the frequency of caching queries. We show that using this architecture, high throughput can be achieved with much smaller buffers (hence much better freshness) than in FCDS.

To lower synchronization overhead, we refrain from synchronizing insertions to the *Gather&Sort* with propagations from it, which may lead to buffered elements being sporadically overwritten by others without being propagated, while others may be duplicated, i.e., propagated more than once. These occurrences, which we call *holes*, alter the stream ingested by the data structure. Yet, we show that for a sufficiently large local buffer, the expected number of holes is small and, because they are random, they do not change the sampled distribution. Figure 4.2 presents quantiles estimated by Quancurrent on a stream of normally distributed random values compared to an exact, brute-force computation of the quantiles, and shows that the estimation remains accurate.

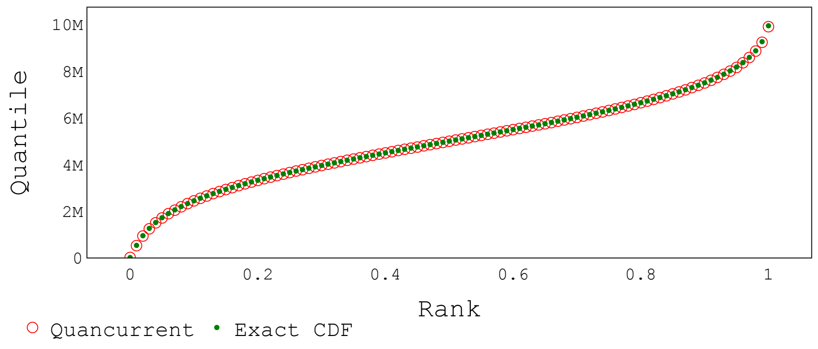
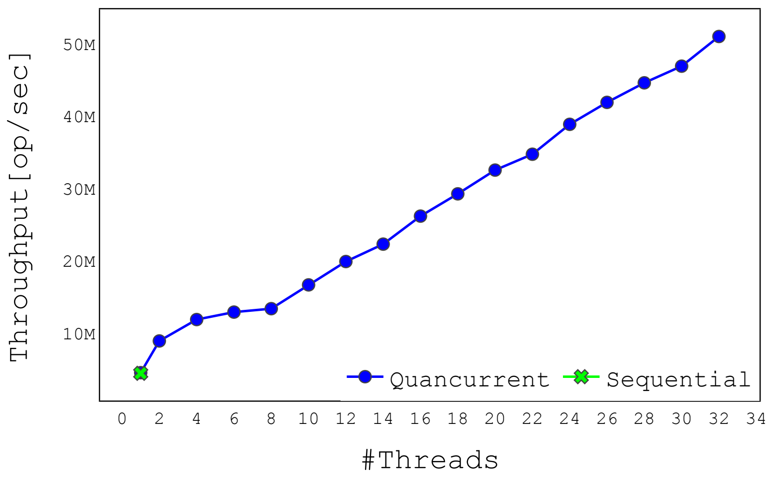
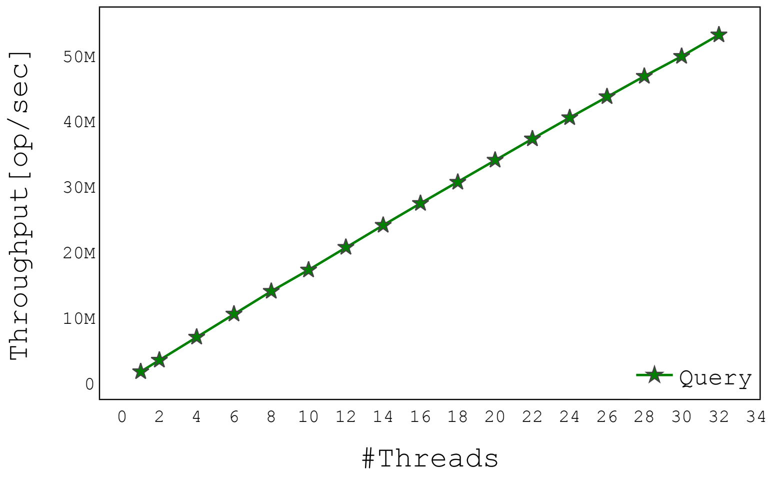


Figure 4.2: Quancurrent quantiles vs. exact CDF, k = 1024, normal distribution, 32 update threads, 10M elements.

We achieve an update speedup of 12x and a query speedup of 30x over the sequential sketch, both with linear speedup, as shown in Figure 4.3. In the full paper [41], we compare Quancurrent to FCDS, which is the state-of-the-art in concurrent sketches, and show that for FCDS to achieve similar performance it requires an order of magnitude larger buffers that Quancurrent, reducing query freshness tenfold.



(a) Update-only.



(b) Query-only.

Figure 4.3: Quancurrent Speedup.

Chapter 5

Compressing Distributed Network Sketches with Traffic-Aware Summaries

5.1 Introduction

Network designers need to gather analytics about the performance of the network to better understand what is happening behind the curtain. These are useful for traffic engineering, for reaching a higher utilization of the infrastructure, for reducing link congestion, and for detecting anomalies when they happen. Switches generally do not have sufficient memory to hold entire measurement models for large data streams, therefore the network designer generally sacrifices accuracy for a lower memory footprint. Data sketching algorithms, or *sketches* for short [29], are an indispensable tool for such high-speed low-memory-footprint computations. Sketches estimate some function of a large stream such as flow sizes [31] (i.e., the number of packets in a flow), stream cardinality [33, 50] (i.e., the number of flows in a stream), the top-k most common items [88] or frequency changes [76]. They are supported by many data analytics platforms such as PowerDrill [61], Druid [39], Hillview [62], and Presto [95] as well as by standalone toolkits [118].

Measurements are conducted in switches all over the network; however, to successfully analyze network behavior, measurement data needs to be gathered in a centralized server that can see a wider view. A stream of multiple flows passes through the network, with disjoint parts of the stream sampled by multiple *ingestion nodes*, where parts of the same flow may be ingested by different nodes. The ingestion nodes periodically propagate their local sketch to a *central node*, as illustrated in Figure 1.3. The network has to handle the trade-off of sending data packets vs. sending crucial control packets with sketch information [124, 20]. A way to reduce these control packets is by compressing the sketches before transferring them to a central analytics node.

Compressing is crucial as network managers may have many sketches estimating different measurements concurrently. While each sketch alone may be insignificant, their aggregate is noticeable at the network level. Additionally, the sketch may be collecting measurements in low bandwidth networks [14, 87], making the sketch size critical.

A framework for sketch compression was suggested by Yang et al. [119]. A major component of this framework, named *Maximum Merging Algorithm (**MM**)*, compresses a Count-Min sketch (CM) by utilizing a max function and merging multiple cells in one CM to generate a new, smaller sketch. The first limitation of that approach is that it compresses a sketch only into smaller sketches of particular sizes, those that have a common divisor with the size of the original sketch. Moreover, the approach does not provide insights for the selection of the various compression ratios that should be implied for a distributed measurement in multiple nodes. In particular, how the traffic distribution among the nodes should be considered.

A clear drawback of compression methods is the accuracy reduction they might imply. It is often critical for network managers to have access to measurements with guarantees of their accuracy. In the common case that the network has multiple ingestion nodes that generate the measurements and send the data to a centralized server, it is intuitive to allow each node to report them through an amount of data that is correlative to the amount of traffic it observes. In this paper, *we study distributed sketch-based network measurement with limited communication. For optimizing accuracy we call for traffic-aware compression ratios of the multiple sketch instances.* We present a formal model that refers to any given number of ingestion nodes and any distribution of the traffic through these nodes. Moreover, the resize factors also guarantee that the amount of data sent over the network is less than compressing all the data from all the nodes to the same size with previously presented methods.

We present the following major contributions:

(i) As a building block, we develop a compression method named CM-SKTC for a single CM that allows general compression ratios, and then present the Traffic-Aware CM sketch, denoted TA-CM.

(ii) We present Traffic-Aware K-minimum-values (KMV), denoted TA-KMV – a traffic-aware compression ratio for nodes implementing distributed distinct flow count with the KMV sketch.

(iii) Finally, we present Traffic-Aware HyperLogLog (HLL), denoted TA-HLL – a traffic-aware compression ratio for nodes implementing distributed distinct flow count with the HLL sketch.

The rest of the paper is organized as follows. Section 5.2 discusses preliminaries, our model, and related work. In Section 5.3 we present the CM-SKTC compression method for a single CM, bound its error, and describe how to decrease the data sent from ingestion nodes to the centralized server. The TA-CM for traffic-aware CM compression for multiple nodes is described in Section 5.4. Section 5.5 presents the TA-KMV for cardinality estimation (count distinct). Experimental evaluation of the methods is provided in Section 5.7. Finally, in Section 5.8 we conclude and suggest directions for future work.

5.2 Preliminaries

In this section we present the model, as well as background on sketches. Finally, we detail related work.

5.2.1 Model

We consider the incoming packets in the network as originating from a single *stream*. A *flow* is defined to be a sub-stream with common packet headers, e.g., grouping together all packets containing the same 5-tuple. A *measurement* is some function f computed over the stream, e.g., returning the distinct number of flows in the stream. As streams are generally too large to maintain in memory [28], accuracy is traded off for a lower memory footprint [31, 4, 19] by algorithms called *sketches*. Several sketches have the *mergeability* property that refers to the ability for computing a sketch over a stream by merging sketches over substreams.

As mentioned previously, measurements are conducted in switches all over the network. We model this by splitting the stream into disjoint sub-streams observed by *ingestion nodes*, where parts of the same flow may be observed by different ingestion nodes. Once the nodes have a sketch ready to be sent (e.g., periodically or upon receive a specific signal) they propagate their local sketch or a summary for it to a *central node* [64, 81, 82], which merges all sketches to provide the desired measurement.

5.2.2 Background

In this section, we present the HLL sketch and the Maximum Merging (MM) method.

HyperLogLog for Cardinality Estimation

HyperLogLog (HLL) [49, 61] is another popular algorithm for cardinality estimation. As a building block, it relies on the Flajolet–Martin method of estimating the number of distinct elements [50, 40]. They propose computing a hash value $h(a_i)$ for each element a_i and a value $\rho(h(a_i))$ that indicates the index of the first non-zero bit upon considering the binary representation of $h(a_i)$, starting from the least significant bit. The HLL holds an array M of $m = 2^d$ counters. For an element a_i, a counter is selected based on the d left-most bits in $h(a_i)$ and this counter is potentially updated based on the next t bits in $h(a_i)$. The use of hashing ensures that repeated elements in the stream imply the same values and thus cannot increase the counter values. Finally, the cardinality estimation computed based on the m counters is:

$$\alpha_m \cdot m^2 \cdot (\sum_{j=0}^{m-1} 2^{-M[j]})^{-1}.$$

α_m is approximated as:

$$\alpha_m = \begin{cases} 0.673 & m = 16 \\ 0.697 & m = 32 \\ 0.706 & m = 64 \\ \frac{0.723}{1+\frac{1.079}{m}} & m \geq 128 \end{cases}$$

as shown in [49].

Maximum Merging (MM)

Previous work by Yang et al. has presented the Maximum Merging (**MM**) [119] method, a method for compressing CM before transmission over the network. In this method, the CM sketch is compressed from size w to w', where w' is a divisor of w. Column i is added to column ($i \mod w'$). If $w = r \cdot w'$, the CM sketch is compressed by a factor r earlier to its transmission over the network.

While this method maintains the lookup speed of the original CM sketch, a limitation of this method is the requirement that w' be a divisor of w. Even more important, in the common scenario of multiple measurements, the MM scheme compresses all sketches equally regardless of the amount of traffic each of them observes. We indicate that this scheme can lead to under utilized bandwidth. For example, if the two node processed different portions of the stream, an identical compression implies over-compression of the sketch that observes more traffic, leading to a higher overall error.

5.2.3 Related Work

Common sketch solutions focus on the trade-off of speed, accuracy, and memory (e.g., [31, 28, 88, 107, 44, 96, 97, 102] and more). However, they generally consider only a single node ingesting the entire stream. The introduction of software-defined networks (SDN) allows for deploying centralized algorithms for maintaining and collecting information about network operations [120]. These solutions typically have a centralized controlled merging of incoming data from ingestion nodes. Network-wide measurements have been widely studied [2, 10, 81, 58]. These solutions, however, do not consider the size of control packets sent to the controller – they do not take into account that these control packets may worsen existing congestion [124, 20].

Shrivastava et al.[106] presented time adaptive sketches and discussed the need for having recent data more accessible. In [57], Harrison et al. presented a network-wide scheme of detecting heavy-hitters while considering the reporting communication overhead. [58] presented a method of heavy-hitters detection that used probabilistic summary reporting to decrease control packets during DDoS attacks. These two works emphasize that minimizing summaries size is critical and can have a major influence on network performance. By using methods of our paper, network operators can create larger sketches and excess the need for time-reliant sketches.

Recently, [103] suggested methods for accurate flow size estimation in the Count-Min sketch (CM) without overestimation, that apply when the number of flows of non-zero size is bounded. Yang et al. presented the Maximum Merging (**MM**) [119], a method for compressing CM before transmission over the network. MM allows providing estimations under a range of traffic characteristics implying various bandwidth constraints. The main limitation of this method is that it can only compress the CM to a constant w' which divides the width w. MM does not deal with two aspects that we find important for making it practical. It refers to a single sketch and does not discuss the compression of multiple sketch instances and in particular how to compute various compression ratios for such sketches based on the traffic. Likewise, it is focused solely on the CM. In this work, we study the common scenario of multiple distributed sketches and refer in addition to CM also to other common sketches.

Our methods utilize similar patterns to the original sketches, therefore our proposed sketches and techniques may have the potential to be implemented on programmable switches. For example, the following sketches have been implemented using the Tofino architecture [69]: the CM sketch [91], the Quantiles sketch [93], and more [123, 56, 122].

5.3 The CM-SKTC Compression Method

We present a method that allows compressing any sized CM (d, w) to any new size possible (d, w') for $w' \in [1, w]$. Let $h_1, \ldots, h_d$ be pair-wise independent hash functions used in original CM, such that every function holds $h_i : \{f_1, f_2, \ldots\} \mapsto \{1, 2, \ldots, w\}$, where f_j is flow j. Let $g_1, ..., g_d$ be pair-wise independent hash functions such that $g_i : \{1, 2, \ldots, w\} \mapsto \{1, 2, \ldots, w'\}$ for every i.

The CM-SKTC compression method works as follows. For each array $S^c[j]$ in the sketch, the value of the l's cell $S^c[j][l]$ is the maximum value over all cells in the corresponding array of the original sketch $S[j]$ that by the hash function g_j are hashed to the l'th cell (i.e. $S^c[j][l] \leftarrow \max\limits_{i:g_j(i)=l} \{S[j][i]\}$). Algorithm 7 presents the CM-SKTC compression method pseudo code.

An example of the compression process of CM-SKTC is illustrated in Figure 5.1 where a CM-SKTC of size $d = 2, w' = 4$ is computed for a CM of size $d = 2, w = 6$. Notice that the method reduces the number of columns (rather than that of the rows) since typically the number of rows is low beforehand, as there is a need for a distinct hash function per row.

The query method (Algorithm 8) from CM-SKTC is similar to the original CM

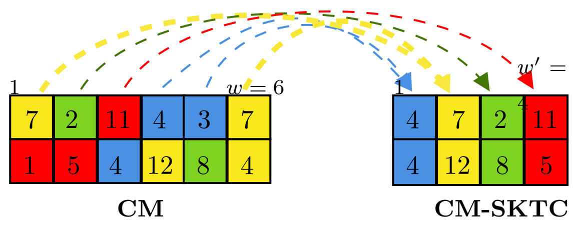


Figure 5.1: An example for Count-Min sketch (CM) (left side) compression with CM-SKTC (right side). In this example, a CM of size $d = 2, w = 6$ is compressed into a CM-SKTC of size $d = 2, w' = 4$. Values of the hash functions g_1, g_2 are represented by various colors. In row i, multiple counters with the same hash value of g_i are represented by their maximum.

query. When flow f estimation value is required, for each array $S^c[j]$ (where $j \in [1, d]$) find the appropriate cell $S^c[j][g_j(h_j(f))]$ (one in each array) and return their respective minimal value.

Similar to the original CM and the **MM** [119], the CM-SKTC method also generates only overestimation values.

Algorithm 7 CM-SKTC compression algorithm

```
   vars:
   $h_1, \dots h_d$ - hash functions from flows to $[1, w]$
   $g_1, \dots g_d$ - hash functions from $[1, w]$ to $[1, w']$
1: procedure COMPRESS($S$)
2:     $S^c \leftarrow$ array of 0's of size $d \times w'$
3:     for $j \in [1 \dots d]$ do
4:         for $i \in [1 \dots w]$ do
5:             $l \leftarrow g_j(i)$
6:             $S^c[j][l] = \max(S^c[j][l], S[j][i])$
7:     return $S^c$
```

Algorithm 8 CM-SKTC estimation query

```
   vars:
   $h_1, \dots h_d$ - hash functions from flows to $[1, w]$
   $g_1, \dots g_d$ - hash functions from $[1, w]$ to $[1, w']$
1: procedure QUERY($S^c, f$)
2:     $est \leftarrow 0$
3:     for $j \in [1 \dots d]$ do
4:         if $est > S^c[j][g_j(h_j(f))]$ then
5:             $est \leftarrow S^c[j][g_j(h_j(f))]$
6:     return $est$
```

Yang et al. [119] present error bounds for compressing the CM when reducing w to some divider w' (i.e., $w = z \cdot w'$ for an integer z). The number of counters compressed to the same counter is fixed as w/w'. Our CM-SKTC, however, maps a variable number of counters to each counter using hashing. This number can vary among the counters in an array or among the arrays, thereby adding a taste of randomness. Recall that in the CM sketch δ (the error probability) is a function of the number of columns and ϵ

(the error) is a function of the number of rows. Our compression scheme reduces the number of columns, therefore, intuitively, the probability of estimating the flow size with error at most ϵ is reduced by some factor. In practice, given a probability δ, the CM-SKTC is initialized with enough columns so that after compression the probability of estimating the flow size with at most error ϵ is greater than $1-\delta$. Given a CM sketch with d rows, and w columns, and a compression ratio w'/w, Lemma 5.3.1 presents the guarantees of the compressed sketch.

Lemma 5.3.1. *Given a CM S with d rows and w columns, for appropriate parameters (ϵ, δ'), and a CM-SKTC compression ratio $\frac{w'}{w} < 1$. Let N be the size of the stream and denote by $\hat{f}_i$ the estimation of flow with size f_i. Denote by β the term*

$$\left(1-\left(1-\frac{1}{\epsilon w}\right)\left(1-\frac{N}{w(f_i+\epsilon N)}\right)^{\frac{w}{w'}-1+\sqrt{-2\ln(1-\delta)\frac{w}{w'}}}\right)^d.$$

The estimation $\hat{f}$ of flow f satisfies $\hat{f} \leq f + \epsilon N$ with probability $1-\beta-\delta'(1-\beta)$.

Proof of Lemma 5.3.1. Following the compression, each counter in array $j \in [1, d]$ of S^c is the maximum of all values of counters from S mapped to this counter by the corresponding function from g_j. Since the d arrays in S^c are independent, we begin by analyzing each array error contribution by itself. For some array, let denote by X_l the number of cells that mapped to cell $l \in [1, w']$ in S^c. As the hash functions map the domain $\{1, 2, \ldots, w\}$ uniformly to the range $\{1, 2, \ldots, w'\}$, X_l is a random variable drawn from a binomial distribution with $p = 1/w'$ and $n = w$. As X_l is binomial, $e \triangleq E(X_l) = \frac{w}{w'}$. We now bound the number of rows in the original CM sketch that map to the rows in the compressed sketch using the Chernoff bound:

$$\begin{aligned}
&\Pr\left(X_l > e + \sqrt{-2\ln(\sqrt[d]{1-\delta})e}\right) \\
&= \Pr\left(X_l > \left(1+\sqrt{-2\ln(\sqrt[d]{1-\delta})/e}\right)e\right) \\
\overset{\text{Chernoff bound}}{\leq}\ & e^{-\frac{\left(\sqrt{-2\ln(\sqrt[d]{1-\delta})/e}\right)^2 r}{2}} = e^{-\frac{-2\ln(\sqrt[d]{1-\delta})e}{2e}} \\
&= e^{\ln(\sqrt[d]{1-\delta})} = \sqrt[d]{1-\delta}.
\end{aligned}$$

Let $C = \left\lceil e + \sqrt{-2\ln(1-\sqrt[d]{1-\delta})e} \right\rceil$. Following the compression, each counter in S^c is the maximum value of at least C counters from S with probability at most $\sqrt[d]{1-\delta}$.

We bound the error for a flow f_i, which is mapped to one counter in S and compressed with at most $C-1$ other counters to a new counter with probability at least $1-\sqrt[d]{1-\delta}$. Denote by $n_1, ..., n_C$ the number of packets which do not belong to flow f_i, but are mapped to the C counters mapped to counter S. From the CM sketch, $\mathrm{C}(n_i) \leq \frac{N}{w}$ for every n_i. Without loss of generality assume that f_i is mapped to n_1.

Conclude that the estimation $\hat{f}_i$ is $\hat{f}_i = \max(n_1 + f_i, n_2, ..., n_C)$ and accordingly

$$\begin{aligned}
&\Pr(\hat{f}_i \geq f_i + \epsilon N) \\
&= \Pr(\max(n_1 + a_i, n_2, \dots, n_C) \geq f_i + \epsilon N) \\
&\leq 1 - \Pr(\max(n_1 + a_i, n_2, \dots, n_C) < f_i + \epsilon N) \\
&= 1 - \Pr(n_1 + f_i < f_i + \epsilon N) \cdot \prod_{j=2}^{C} \Pr(n_j < f_i + \epsilon N) \\
&= 1 - \Pr(n_1 < \epsilon N) \cdot \prod_{j=2}^{C} \Pr(n_j < f_i + \epsilon N)
\end{aligned}$$

$$\begin{aligned}
&\overset{\text{Markov inequality}}{\leq} 1 - \left(1 - \frac{E(n_1)}{\epsilon N}\right) \cdot \prod_{j=2}^{C} \left(1 - \frac{E(n_j)}{f_i + \epsilon N}\right) \\
&\leq 1 - \left(1 - \frac{1}{w\epsilon}\right) \left(1 - \frac{N}{w(f_i + \epsilon N)}\right)^{C-1}
\end{aligned}$$

As the hash functions are pairwise independent, we can use this bound to bound the error of $\hat{f}_i$, the estimation of flow f_i. By definition $\hat{f}_i = \min(\hat{f}_i^1, \hat{f}_i^2, \dots, \hat{f}_i^d)$.

$$\begin{aligned}
&\Pr(\hat{f}_i \geq f_i + \epsilon N) = \Pr(\min(\hat{f}_i^1, \dots, \hat{f}_i^d) \geq f_i + \epsilon N) \\
&\leq \prod_{j=1}^{d} \Pr(\hat{f}_i^j \geq f_i + \epsilon N)
\end{aligned}$$

We have shown that for any j: $\Pr(\hat{f}_i^j \geq f_i + \epsilon N) \leq 1 - \left(1 - \frac{1}{w\epsilon}\right) \left(1 - \frac{N}{w(f_i+\epsilon N)}\right)^{C-1}$, with probability $\delta' = \sqrt[d]{1-\delta}$. Therefore, as the hash functions as pair-wise independent, we can conclude that the following holds with probability of $\sqrt[d]{1-\delta}^d = 1 - \delta$:

$$\begin{aligned}
&\Pr(\hat{f}_i \geq f_i + \epsilon N) \leq \prod_{j=1}^{d} \Pr(\hat{f}_i^j \geq f_i + \epsilon N) \\
&\leq \prod_{j=1}^{d} 1 - \left(1 - \frac{1}{w\epsilon}\right) \left(1 - \frac{N}{w(f_i + \epsilon N)}\right)^{C-1} \\
&= \left(1 - \left(1 - \frac{1}{w\epsilon}\right) \left(1 - \frac{N}{w(f_i + \epsilon N)}\right)^{C-1}\right)^d = \beta.
\end{aligned}$$

With probability of $1 - \delta$ the following holds: $\Pr(\hat{f}_i \geq f_i + \epsilon N) \leq \beta$, thus $\Pr(\hat{f}_i \leq f_i + \epsilon N) \geq 1 - \beta$ with probability at least $1 - \delta$. Therefore the probability that $\hat{f}_i \leq f_i + \epsilon N$ is at least $(1 - \delta)(1 - \beta) = 1 - \beta - \delta(1 - \beta)$. $\square$

Lemma 5.3.1 shows that compressing a sketch with parameter δ yields a sketch with parameter $\beta + \delta(1 - \beta)$, where β is as defined in the lemma. It follows that to achieve probability δ **after** compression, we must instantiate the uncompressed sketch with parameter $(\delta-\beta)/(1-\beta)$. Denote $\delta' = \beta + \delta(1-\beta)$. We empirically analyze how the compression affects the probability. Table 5.1 shows how δ' is affected by changes in δ, w'/w, and ϵ respectively. δ' remains less than 3δ, therefore we initialize the CM sketch to have the number of columns for 3δ as opposed to δ. As the number of columns is $\lceil \ln 1/\delta \rceil$, generally, we need to add a single column in order for the error probability after compression to be less than δ.

5.4 The Traffic-Aware CM

The system measures a stream of data that is split in a distributed fashion across several ingestion nodes. Upon stream ingestion completion, the nodes communicate with a centralized server which can then answer queries. This is depicted in Figure 1.3. Queries are agnostic to the specific architecture, namely, they only refer to the complete stream as a whole and do not refer to its parts observed by each of the nodes. There is a clear tradeoff between the summaries size sent to the server and its ability to answer queries accurately. Accordingly, we would like to optimally use compression to allow high accuracy. The **MM** [119] scheme was proposed to reduce summary size through compression of CMs maintained by the nodes but the fact it can be compressed in only particular ratios restricts it from taking advantage of all bandwidth to the server.

In network traffic, flow size distribution among switches can be imbalanced [72, 104]. For instance, the location of nodes along paths of various lengths or employment of particular network functions in the nodes can result in nodes receiving a small portion of the network stream while others more traffic. We leverage this skew to reduce the summaries size sent to the server. Specifically, we aim to compress the sketches sent by ingestion nodes that received a small portion of the overall stream more than those sketches of nodes with higher portions of traffic. We say that a sketch compression from w columns to w' columns has a *compression ratio* of w'/w. As mentioned, the number of rows is not reduced.

The design of Traffic-Aware Count-Min Sketch (TA-CM) is as follows: Given parameters (ϵ, δ), where ϵ is the desired error, δ is the maximum error probability, we instantiate a CM on every ingestion node with parameters $(\sigma \cdot \epsilon, \delta)$ for some $0 < \sigma \leq 1$. $\frac{1}{\sigma}$ is an enlargement factor and is known in advance to the ingestion nodes and the centralized server. We increase the size of the sketch at the ingestion nodes by $\frac{1}{\sigma}$ and as

Table 5.1: Error probabilities after compression, for values of δ, ϵ and the resize factor, as defined by Lemma 5.3.1.

Resize Factor	0.6		0.9	
δ \ ϵ	0.02	0.06	0.02	0.06
0.04	0.0936	0.1177	0.0841	0.0925
0.08	0.1791	0.217	0.1648	0.1808

such decrease their respective error (as the error is tied to the number of columns). By enlarging the sketches at the ingestion nodes, we can compress them while maintaining an error within the desired bounds. When the ingestion period ends, the following protocol takes place:

(i) Each ingestion node reports to the central node its local stream size.

(ii) The centralized server computes each ingestion node's compression ratio.

(iii) Every ingestion node compresses its CM with the CM-SKTC method, then sends its summary to the centralized server.

(iv) The centralized node can query a single flow f_j by querying every CM-SKTC separately and summing up the results to calculate the estimation for flow f_j.

To achieve an overall maximum error of ϵ for an arbitrary flow size estimation, the centralized server can (naively) request a compression ratio of $1/\sigma$. Theorem 5.1 shows that when there are two ingestion nodes, there are optimal compression ratios that minimize the total summaries size while maintaining the error bounds of the flow size estimation in the centralized server to be at most ϵ.

Intuitively, the proof is based on constraints that: (i) the overall error be within the desired bounds, and (ii) the size of sent traffic be smaller than the naive solution (i.e., less than $2 \cdot w \cdot d$ where $w \cdot d$ is the size of a CM with parameters (ϵ, δ)).

Theorem 5.1. *For given Count-Min sketch parameters ϵ, δ, initial resize factor σ, and two CMs S_1, S_2, with stream sizes of N_1, N_2 respectively, such that w.l.o.g $N_1 \geq N_2$. The TA-CM resize factors $r_1 = \frac{k+1}{\sigma(k+\sqrt{k})}$ and $r_2 = \frac{k+1}{\sigma(\sqrt{k}+1)}$ for $k = N_1/N_2$ generate the minimal amount of network communication such that the error is bounded by ϵ with probability at least $1-(\beta+\delta(1-\beta))$.*

Proof of Theorem 5.1. Denote by f the flow whose size is being estimated and by $\hat{f}$ the estimation. Given a CM with parameters $(\sigma\epsilon, \delta)$, after compression by a factor of r, the following inequality holds with probability at least $1-(\beta+\delta(1-\beta))$, by Lemma 5.3.1: $f \leq \hat{f} \leq f + \sigma\epsilon r N$ Denote f_1, f_2 the flow ingested by S_1, S_2 respectively, and, likewise, their estimations (after compression) by $\hat{f}_1$, $\hat{f}_2$.

Recall that: $f_1 \leq \hat{f}_1 \leq f_1 + \sigma\epsilon r_1 N_1$, and $f_1 \leq \hat{f}_2 \leq f_2 + \sigma\epsilon r_2 N_2$, where $f = f_1 + f_2$. Therefore, due to the mergeability property, the estimation by the central entity is:

$$f \leq \hat{f}_1 + \hat{f}_2 \leq f + \sigma\epsilon(r_1 N_1 + r_2 N_2)$$
$$f_1 + f_2 \leq \hat{f}_1 + \hat{f}_2 \leq f_1 + \sigma\epsilon r_1 N_1 + f_2 + \sigma\epsilon r_2 N_2$$

Recall that we would like to maintain an overall error of ϵ, therefore we require that:

$$\sigma(r_1 N_1 + r_2 N_2) \leq N,$$

and that $N_1 = kN_2$. Denote the ratio between r_1 and r_2 as $\alpha = r_1/r_2$. As N_1 is larger than N_2, we expect α to be less than 1, as the information retained in S_2 will have less of an impact than S_1:

$$\begin{aligned}
&N_1 + N_2 = N \Rightarrow N_2 = \frac{N}{k+1}\\
&\sigma(r_1N_1 + r_2N_2) = \sigma(\alpha k r_2 N_2 + r_2 N_2) \le N\\
&r_2 \le \frac{k+1}{\sigma(\alpha k + 1)}
\end{aligned}$$

To reduce the total summaries size as much as possible, the ratios r_1 and r_2 should be as large as possible:

$$r_2 = \frac{k+1}{\sigma(\alpha k + 1)} \tag{5.1}$$

The goal is to reduce the total summaries size in comparison to trivial resizing by a factor of $1/\sigma$. We require:

$$\frac{1}{r_1} + \frac{1}{r_2} \le 2 \cdot \frac{1}{1/\sigma} = 2\sigma \tag{5.2}$$

To achieve minimum summaries size, we minimize the left-hand size of Equation (5.2). Recall that $r_1 = \alpha r_2$ for some $\alpha \in (0, \infty)$.

$$\begin{aligned}
&\min_{\alpha\in(0,\infty)} \left\{\frac{1}{r_1} + \frac{1}{r_2}\right\} = \min_{\alpha\in(0,\infty)} \left\{\frac{1}{\alpha r_2} + \frac{1}{r_2}\right\} =\\
&\min_{\alpha\in(0,\infty)} \left\{\frac{1+\alpha}{\alpha r_2}\right\} \overset{(i)}{=} \min_{\alpha\in(0,\infty)} \left\{\sigma(\alpha k + 1)\frac{1+\alpha}{\alpha(k+1)}\right\} \overset{(ii)}{=}\\
&\min_{\alpha\in(0,\infty)} \left\{\alpha k + \frac{1}{\alpha}\right\}
\end{aligned}$$

Where (i) is from Equation (5.1) and (ii) is by removing constants not affected by α. The minimum is for $\alpha = 1/\sqrt{k}$, implying $\alpha < 1$ for $k > 1$, as expected. Finally, our last constraint is that merging the sketches by factors other than $1/\sigma$ should provide smaller summaries size:

$$\frac{1}{r_1} + \frac{1}{r_2} = \sigma(\alpha k + 1)\frac{1+\alpha}{\alpha(k+1)} \overset{\alpha=1/\sqrt{k}}{=} \sigma\frac{(\sqrt{k}+1)^2}{k+1}.$$

□

The theorem shows that, as the imbalance in ingested stream parts between the nodes (k) increases, sketch S_2 is compressed to a higher degree. Furthermore, at the extreme case (i.e., $k = \infty$), S_1 is compressed by a factor of $1/\sigma$, whereas S_2 is compressed by a factor of ∞. At this extremity, S_2 contains no information, thus this compression is intuitive. Furthermore, when $k = 1$ both sketches are compressed by a ratio of $1/\sigma$, which is also intuitive.

We now generalize Theorem 5.1 to the practical case of an arbitrary number of n

nodes. We again maintain error bounds within certain limits. We first prove a helpful lemma:

Lemma 5.4.1. *For $n \geq 2$ the variables $1 \leq y_1, .., y_n$ satisfy*

$$\frac{\left(1+\sum\limits_{i=1}^{n}\prod\limits_{j=i}^{n}\sqrt{y_j}\right)^2}{\left(1+\sum\limits_{i=1}^{n}\prod\limits_{j=i}^{n}y_j\right)} \leq n+1.$$

Proof of Lemma 5.4.1. Let $y_1, \ldots, y_n$ be as required. Denote:

$$x_i \triangleq \prod_{j=i}^{n} y_j.$$

Note that, as $1 \leq y_1 \ldots y_n$, then $1 \leq x_i$ for all i.

As

$$\left(1+\sum_{i=1}^{n}\sqrt{x_i}\right)^2 = 1+\sum_{i=1}^{n}x_i+\sum_{i=1}^{n}2\sqrt{x_i}+\sum_{i\neq j}\sqrt{x_i x_j},$$

then

$$\frac{\left(1+\sum\limits_{i=1}^{n}\sqrt{x_i}\right)^2}{1+\sum\limits_{i=1}^{n}x_i} = 1+\frac{\sum\limits_{i=1}^{n}2\sqrt{x_i}+\sum\limits_{i\neq j}\sqrt{x_i x_j}}{1+\sum\limits_{i=1}^{n}x_i}$$

so we need to prove that

$$\frac{\sum\limits_{i=1}^{n}2\sqrt{x_i}+\sum\limits_{i\neq j}\sqrt{x_i x_j}}{1+\sum\limits_{i=1}^{n}x_i} \leq n$$

Using the arithmetic-mean–geometric-mean inequality:

$$\sum_{i\neq j}\sqrt{x_i x_j} \leq \sum_{i\neq j}\frac{x_i+x_j}{2} = \sum_{i=1}^{n}(n-1)x_i.$$

The divisor disappears as we double count every x_i: once on the left hand of the + and once on the right hand. Furthermore, for any $x_i \geq 1$:

$$2\sqrt{x_i}+(n-1)x_i \leq 1+n\cdot x_i.$$

Putting these together we get:

$$\sum_{i=1}^{n}2\sqrt{x_i}+\sum_{i\neq j}\sqrt{x_i x_j} \leq \sum_{i=1}^{n}2\sqrt{x_i}+\sum_{i=1}^{n}(n-1)x_i =$$

$$\sum_{i=1}^{n}\left(2\sqrt{x_i}+(n-1)\,x_i\right) \leq \sum_{i=1}^{n}\left(1+n\cdot x_i\right) = n+n\sum_{i=1}^{n}x_i$$

Finally:

$$\frac{\sum\limits_{i=1}^{n} 2\sqrt{x_i} + \sum\limits_{i\neq j} \sqrt{x_i x_j}}{1 + \sum\limits_{i=1}^{n} x_i} \leq \frac{n + n\sum\limits_{i=1}^{n} x_i}{1 + \sum\limits_{i=1}^{n} x_i} = n$$

□

We now use the lemma to prove the following theorem:

Theorem 5.2. *Given $n \geq 2$ CMs S_i, with stream sizes $N_1 \geq N_2 \ldots \geq N_n$. Denote $k_i = N_i/N_{i+1}$ for $i \in [1, n-1]$ and $k_n = 1$. The optimal TA-CM resize factors are*

$$\forall i \in [1, n-1] : \; r_i = \frac{r_{i+1}}{\sqrt{k_i}} \quad \textit{and} \quad r_n = \frac{\sum\limits_{i=1}^{n} \prod\limits_{j=i}^{n} k_j}{\sigma(\sum\limits_{i=1}^{n} \prod\limits_{j=i}^{n} \sqrt{k_j})}.$$

Proof of Theorem 5.2. The proof follows a similar path as the proof for Theorem 5.1. Denote by f_i the size of the stream i observed by S_i, and the estimations by $\hat{f}_i$. Denote by r_i the compression ratio of S_i. Let δ' be the probability calculated for some compression rate r, to be determined. Using the mergeability property, the estimation by the central entity, with probability at least $1 - \delta'$, is:

$$f \leq \sum_i \hat{f}_i \leq f + \sigma\epsilon \sum_i r_i N_i$$

$$\sum_i f_i \leq \sum_i \hat{f}_i \leq \sum_i (f_i + \sigma\epsilon r_i N_i)$$

Recall that we wish to maintain an overall bound of ϵ, thus:

$$\sigma\epsilon \sum_i r_i N_i \leq \epsilon N$$

Denote $k_i = \frac{N_i}{N_{i+1}}$, and $\alpha_i = \frac{r_i}{r_{i+1}}$. Therefore:

$$\sigma\epsilon r_n N_n \sum_{i=1}^{n} \prod_{j=i}^{n} \alpha_j k_j \leq \epsilon N$$

Since $N = N_1 + N_2 + ... + N_n$ and $N_i = N_n \cdot \prod\limits_{j=i}^{n} k_j$, similar to the proof for Theorem 5.1, observe that

$$r_n \leq \left(\sum_{i=1}^{n} \prod_{j=i}^{n} k_j \right) / \left(\sigma \cdot \sum_{i=1}^{n} \prod_{j=i}^{n} \alpha_j k_j \right).$$

We maximize r_n to minimize summaries size. We choose minimal such α_i values by finding the minimum of $\min_{\alpha_i} \left\{ \sum_{i=1}^{n} \frac{1}{r_i} \right\}$.

Following the proof for Theorem 5.1 we use $\alpha_i = \frac{1}{\sqrt{k_i}}$ brings the summaries size to less than the trivial compression [1]. Therefore, we reduce the size if the following holds:

$$\left(1 + \sum_{i=1}^{n-1} \prod_{j=i}^{n-1} \sqrt{k_j}\right)^2 / \left(1 + \sum_{i=1}^{n-1} \prod_{j=i}^{n-1} k_j\right) \leq (n-1) + 1$$

By Lemma 5.4.1 this expression is true for all ratios $k_i \geq 1$ and the resize factors are given by:

$$r_n = \left(\sum_{i=1}^{n} \prod_{j=i}^{n} k_j\right) / \left(\sigma \cdot \sum_{i=1}^{n} \prod_{j=i}^{n} \sqrt{k_j}\right),$$

implying $r_i = \frac{r_{i+1}}{\sqrt{k_i}}$. Finally, using r_n to calculate δ' (i.e., taking the lowest probability), we have proven our theorem. □

Theorem 5.2 shows that sketches that have ingested a larger portion of the overall traffic are compressed less. This is similar to the case of 2 nodes, as nodes that have ingested less traffic have a smaller impact on the query. Note that if all nodes ingest an equal part of the stream, i.e., $k_i = 1$ for all i, then the compression ratio is $1/\sigma$ as expected.

5.5 The Traffic-Aware KMV for Cardinality Estimation

We now present Traffic-Aware KMV (*TA-KMV*), a method to compress the KMV sketch [52, 19], a known tool for computing the number of distinct flows in a stream, also known as its cardinality. For the case of a single node KMV works as follows: For every flow $f_1, f_2, \ldots$ it generates hash values $h_1, h_2, \ldots$ and it retains the k smallest values - $\{h'_1, h'_2, \ldots, h'_k\}$. The cardinality estimation the sketch calculates is $\frac{k}{\max\limits_{i\in[1,k]}\{h'_i\}}$. Errors can include overestimations or underestimations of the cardinality and the relative standard error is $\frac{1}{\sqrt{k}}$.

Also for this sketch we refer to the scenario presented in Figure 1.3, with nodes sending summaries to a centralized server through a channel with bounded bandwidth. As a simple baseline solution, every node can compute the hash values for all the flows it observes and send the k minimal values among them to the centralized node. This results in a total number of $n \cdot k$ values sent over the network. Then, the central node simply orders all the $n \cdot k$ values it receives for computing the value which is the k-th smallest in its size among them, denoted as h'_k. It estimates the cardinality of the complete stream as $\frac{k}{h'_k}$. Note that although not all observed values were shared by the various nodes, the value of h'_k necessarily equals the k-th smallest hash value among all the values for the complete stream.

We aim to develop solution that reduces the total number of values that are sent for better utilizing the available bandwidth.

[1] We also conjecture that it is a local minima.

Our proposal for this distributed sketch is as follows: Given parameter k, we instantiate *KMV sketch* on the ingestion nodes with parameter $k \ln k$. When the ingestion ends, the following communication takes place:

(i) Ingestion nodes report their cardinality estimation size to the centralized server.

(ii) The central server computes for each ingestion node i its compression ratio n_i.

(iii) Each ingestion node sends its n_i minimal values to the centralized server.

We suggest a heuristic of calculating compression ratios n_i. We are unable to provide closed form error bounds for this heuristic. Instead, in Section 5.7, we show that this method performs well in realistic scenarios.

The goal is for the centralized server to receive some group of k flow hashes such that the group has as large overlap as possible with the group of global k minimal flow hashes. The method works as follows: Let e_i be the cardinality estimation of server i. First, each ingestion node i sends e_i to the central server, and receives the total estimation $\sum_{j=1}^{n} e_j$ in return. Each ingestion node i can then compute the estimation ratio $r_i = \frac{e_i}{\sum_{j=1}^{n} e_j}$ and sends a summary of size $s_i = r_i \cdot k \ln k$. In this case the total summaries size is:

$$\sum_{i=1}^{n} s_i = \sum_{i=1}^{n} r_i \cdot k \ln k = k \ln k \sum_{i=1}^{n} \frac{e_i}{\sum_{j=1}^{n} e_j} = k \ln k,$$

and accordingly the total size of sent data is $k \ln k + 2n$.

The reason we deliberately suggest sending more then k values is that multiple ingestion nodes may observe traffic from the same flow, and hash values may repeat in the values sent to the centralized node. In such cases among the centralized node will not receive sufficient distinct hash values to achieve the necessary error bounds.

We suggest sending $O(k \ln k)$ hash values, as we observe a similarity between the *TA-KMV* and the coupon collector problem [45, 22]. In the *TA-KMV* the minimal k hash values correspond with coupons, and the goal of the centralized node is to gather these k values in order to achieve a result with an error that is similar to a single node processing the whole stream. While we have no closed-form bounds, the evaluation shows that this method performs well in practice. Note that the coupon collector problem assumes each coupon is drawn independently, which is not the case in our scenario. E.g., the smallest hash may come from a very large stream that appears in every node – in this case, the smallest hash in each node is not independent.

5.6 The Traffic-Aware HLL for Cardinality Estimation

Consider the network-wide measurement framework as illustrated in Figure 1.3 with a channel of limited bandwidth between the nodes to the centralized server and queries on the complete network stream. There can be some overlap between the traffic observed by different nodes thus the number of distinct flows in the stream is not necessarily given by the sum of the distinct flows in each of the nodes $1, \dots, n$.

To estimate the cardinality of the complete stream a HLL can be used. Assume the HLL sketches by each node make use of arrays $M_1, \dots M_n$ of the same size m, using the same set of hash functions. Following its update mechanism as detailed in Section 5.2.2, values of the array M for the complete stream can be computed simply as $M[j] = \max_{i\in[1,n]} M_i[j]$. Accordingly, the HLL array can be computed by the centralized node when each node periodically reports all its array values, allowing the centralized node to estimate the number of distinct elements in the complete stream based on M as $\alpha_m \cdot m^2 \cdot \big(\sum_{j=0}^{m-1} 2^{-M[j]}\big)^{-1}$. Such an approach is communication intensive.

Communication can be saved by reducing the size of the reports sent by the nodes. As $M[j]$ is set to be the maximal among the values $M_1[j], \dots, M_n[j]$, the value $M[j]$ can be computed correctly also when all values besides the maximum are not reported. This motivates a compression scheme in which a node reports an array value based on the probability of the value to be the maximal of the corresponding values (over the nodes) with the same array index. Consider a node i with its j^{th} counter value $c = M_i[j] \geq 0$. Denote N_i as the number of flows observed by node i, and denote $N = \sum_{i=1}^{n} N_i$. In the following lemma we bound from below the probability of the counter j in ingestion node i, to be the maximal among corresponding counters in all the network ingestion nodes.

Lemma 5.6.1. *Let the value of $M_i[j]$ be some $c \geq 0$. The probability that c is the maximal among $M_1[j], \dots, M_n[j]$ is at least $(1 - \frac{1}{m} \cdot 2^{-(c-1)})^{N-N_i}$.*

Proof. Let c be the value of counter $M_i[j]$ for some node i. Consider some flow a observed by some other node k. As the hash function is uniform, it is mapped to counter j with probability $1/m$. By the uniformity of h, with probability $1 - 2^{-(c-1)}$: $\rho(h(a)) \leq c$. As there are at most $N - N_i$ flows observed by all other nodes (as there may be overlap), and as the hash function is pairwise independent, $M_i[j]$ is maximal with probability at least $(1 - \frac{1}{m} \cdot 2^{-(c-1)})^{N-N_i}$. □

As expected, it can be concluded from the bound in Lemma 5.6.1 that as c increases, the probability that the value c is the maximal value among all ingestion nodes increases as well. Note that the values $N_1, N_2, \dots, N_{i-1}, N_{i+1}, \dots, N_n$ might not be known to node i. They can be approximated through a (relatively simple) communication process with the central server such that each node declares its estimation and the central server declares the sum of them all. Without such iteration with the central server, a node can simply report a subset that refers to its largest values. Computing the subset size should consider the number of ingestion nodes.

Consider a family of compression schemes by which each node simply reports a subset of its values. Intuitively, the probability p_c should serve as an input to the decision whether to report the value c. We now detail several potential traffic-aware approaches that can be designed.

(i) (Random Sampling) Select a probability p, fixed for the n nodes based on the limitation of the channel to the centralized server. Each node reports each of its counters w.p. p.

(ii) (Weighted Sampling) A counter value c is reported w.p. p_c or w.p. derived as an increasing function of p_c.

(iii) (Largest Values) Each node can report its $\beta \cdot m$ largest counters, where $\beta \leq 1$ is a small constant. Here, each node reports a similar amount of counters.

(iv) (Threshold) Each node reports all its counters with values of at least a threshold c_T, ignoring small values. The same value c_T applies for all nodes. Hence, the number of values reported by nodes varies based on the values in M_i for a node i. To compute a value c_T that implies reporting some percentage β of the total $n \cdot m$ values, the complete set of $n \cdot m$ values in the n arrays are required. Reporting all such values to the centralized server increases communication costs.

The central server collects the reported values and computes M based on the maximal collected value in each array index $j \in [1, m]$. If for some index, no values were reported they can be set as 0 or as another default value.

5.7 Evaluation

In this section, we evaluate the error bounds of the TA-CM, TA-KMV, and TA-HLL compression methods. For our tests, we use two traces. First, the CAIDA Anonymized Internet Trace 2018 [114], which is a trace collected from the "equinix-nyc" monitor. We also use the MAWILab (MAWI) dataset [100], which contains items for evaluating traffic anomaly detection methods. We implement our algorithms in Python – the source code has been open-sourced [109].

As shown in Section 5.4, the TA-CM sketch has closed-form mathematical guarantees. To this end, we use only the CAIDA dataset to show the error in practice; the analyzed error will hold for any stream. The TA-KMV and TA-HLL sketches have a heuristic approach. We, therefore, use both the CAIDA and the MAWI datasets to evaluate them. We divide the first 50M packets in each trace into batches of 1M packets, resulting in 50 batches per trace. We extract the source-destination pair from every packet and use the pair as the stream elements. Each unique source-destination pair is a flow. Every batch of that CAIDA dataset has $\sim$ 15K flows of varying sizes, while every batch of the MAWI dataset has $\sim$ 100K flows. Each data-point is the average over all batches with each batch being run four times with different hash functions.

The section proceeds as follows: In Section 5.7.1 we analyze the average relative error of the CM-SKTC induced by the compression, compared to the CM sketch and MM compression. Then, in Section 5.7.2, we showcase how we can utilize the CM-SKTC to achieve low error while sending less data over the network for 2 nodes, and in Section 5.7.3 for n nodes. Finally, in Section 5.7.4 we analyze the TA-KMV, and in Section 5.7.5 we analyze the TA-HLL.

5.7.1 Comparing CM-SKTC vs Maximum Merging Algorithm

First, we compare the CM-SKTC to two different compression techniques. (1) No compression, i.e., a CM sketch with $d = 4$, as recommended in [9]. We use varying total memory usage, where each cell is 4 bytes. (2) Maximum Merging (MM) of [119] with an initial 2 MB CM sketch. The evaluation metric we used is *Average Relative Error (ARE)*, defined for a set of flows $\{f_1, ..., f_n\}$ as $\frac{1}{n}\sum_{i=1}^{n}\frac{|\hat{f}_i - f_i|}{f_i} = \frac{1}{n}\sum_{i=1}^{n}\frac{\hat{f}_i - f_i}{f_i}$ where $\hat{f}_i$ is the estimated value of flow f_i.

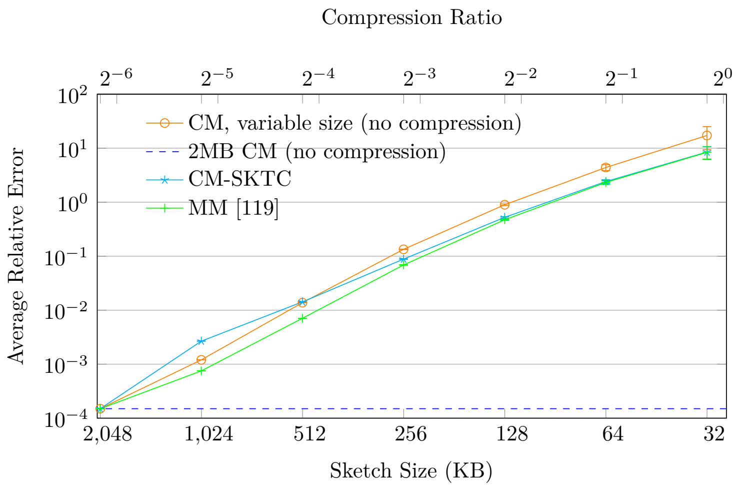


Figure 5.2: Single node: Compression methods comparison

Figure 5.2 compares the different compression methods for different sketch sizes. The blue-dashed baseline represents the *ARE* of the 2 MB CM sketch that both compression methods initiate from. One can observe that CM-SKTC outperforms the two other methods consistently. Furthermore, as the compression ratio decreases and the summary size increases, the *ARE* improves, as expected.

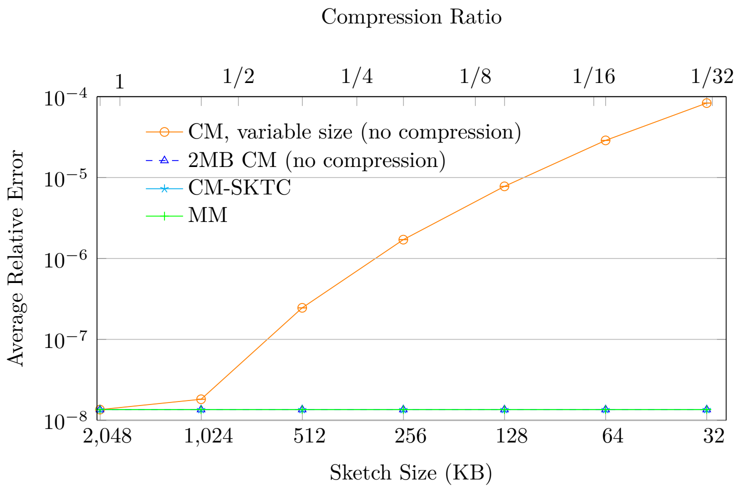


Figure 5.3: Single node: Compression methods comparison - top 50 flows. The 2MB-CM, CM-SKTC, and MM lines coalesce.

Table 5.2: Compression Ratio by k

Compression Method	**Summaries Size Sent (% from max)**
Maximum Merging	2 MB (100%)
TA-CM, $k = 1$	2 MB (100%)
TA-CM, $k = 9$	1.6 MB (80%)
TA-CM, $k = 49$	1.28 MB (64.3%)
TA-CM, $k = 100$	1.18 MB (59%)

In Figure 5.3 we depict the same comparison as Figure 5.2; however we choose a larger sample size of flows, and only to the 50 largest flows ARE in each trace are considered. In this graph, one can observe that the top flows ARE is extremely low for all methods. Moreover, the MM and CM-SKTC curves are similar, and both have the same error as the 2MB sketch.

Note that, the MM is more time-efficient than the CM-SKTC compression. This is due to each hash being calculated twice (for the insertion and for the compression), and therefore we expect the CM-SKTC to be roughly twice as slow as the MM. Table 5.2 shows that our compression scheme has a lower packet overhead than MM. While MM may be faster, we can reduce the bandwidth used by the sketch significantly.

From Figures 5.2 and 5.3, we deduce that the CM-SKTC has two important traits: (1) CM-SKTC achieves estimations within the required error parameters using smaller summaries, and (2) For large (elephant) flows this error is negligible.

5.7.2 TA-CM with two ingestion nodes ($n = 2$)

We now simulate two local nodes receiving a data stream of size N, where the relation between the size of the data stream N_1 processed at the first node, and the size of the data stream N_2 at the second node is $k = \frac{N_1}{N_2}$. We evaluate the effect of k on the ARE.

Figure 5.4 depicts the ARE of merging two sketches when sending different sizes over the network. We compare locally building two sketches with error ϵ, such that they each have size 1MB (meaning 2MB of data is sent over the network), and building larger local sketches with error $\sigma\epsilon$ and compressing them. We compare the trivial compression by factor $1/\sigma$ compared to using our optimal resize factors, for $\sqrt{k} = 3, 7, 10$. Note that our comparison shows that the error of resizing using optimal factors falls in between the error of starting with error ϵ and trivially compressing with error $\sigma\epsilon$. Of great importance is that even in the worst case the error is less than ϵ. Table 5.2 compares the summaries size across the network. It follows that there is a trade-off between the accuracy and the summaries size. Figure 5.5 shows the ratio between summaries size as a function of k, in relation to trivial compression.

In Figure 5.6 we show the results of compressing the same base CM sketch as in Figure 5.4. However, in this simulation, we compare the results when the total summaries size is 2MB, i.e., the summaries account for 2MB of network traffic. In

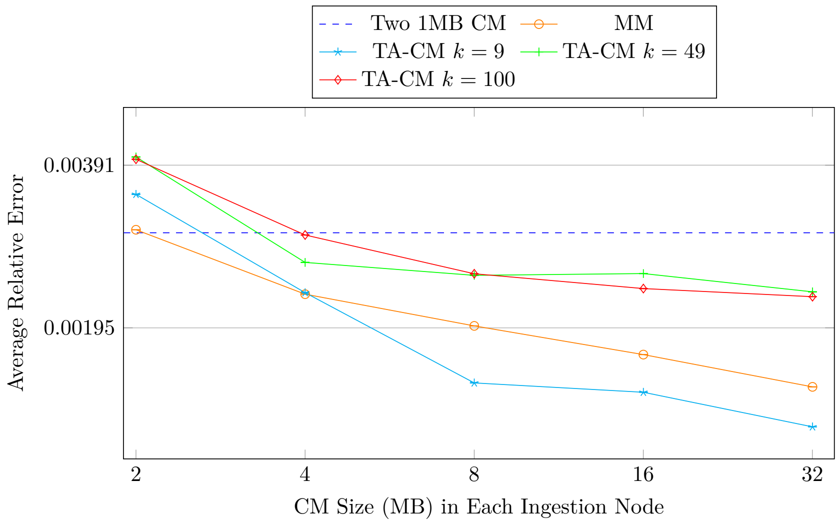


Figure 5.4: Two nodes: Compression methods comparison

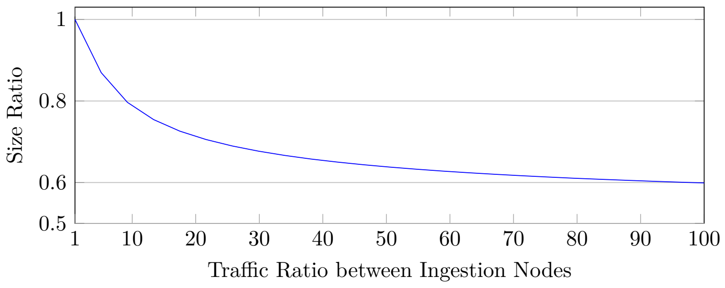


Figure 5.5: Ratio between summaries size in multiple compression methods

this case, we observe that the TA-CM ARE is better than the MM ARE. TA-CM outperforms MM as it is traffic-aware and considers the distribution across the nodes and calculates the ratios accordingly; it helps to send the larger part of the data from the node that handled the larger chunk of the stream.

5.7.3 TA-CM with n ingestion nodes

To evaluate TA-CM in multiple-ingestion nodes scenarios, we formulate four different types of distributions for 10 nodes (see Figure 5.7) and compare the TA-CM to MM and the non-compressed CM sketch. The distributions were chosen such that the ratio between the largest server ratio to the smallest server ratio is 5 (i.e., $N_1/N_{10} = 5$).

The CM sketch base size for each of the 10 nodes is 32KB. We compare the ARE with multiple values of σ (i.e., the ratio by which the ingestion node CM sizes is increased) and compressing with two methods: (1) TA-CM with ratios computed in Section 5.4 (2) MM compression with all ratios are $1/\sigma$. As depicted in Figure 5.8, TA-CM achieves similar results in terms of ARE to the Maximum Merging compression and improves the results of the basic non-compressed CM. However, it does so while decreasing the total summaries size. Table 5.3 indicates that the TA-CM saves between 7% to 9% of

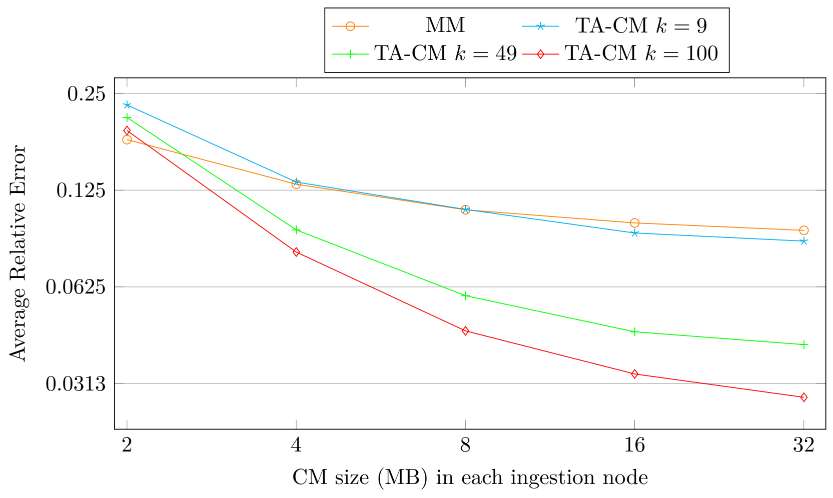


Figure 5.6: Two nodes: Compression methods comparison. Allowance of 2MB sent from ingestion nodes to centralized server.

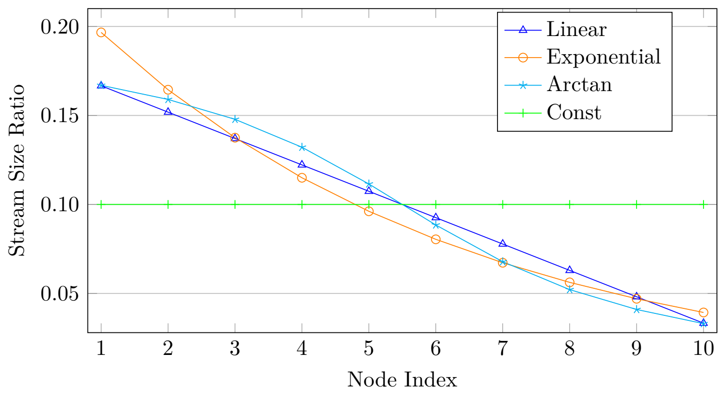


Figure 5.7: $n = 10$ nodes: Stream size distribution over the nodes

the total summaries size for chosen distributions. This saving ratio can be increased by choosing other, wider distributions of the stream (for example if the stream distributes across the ingestion nodes by Pareto distribution then TA-CM potentially saves an even higher percentage).

Table 5.3: Compression ratio of various distributions

Distribution	**Summaries Size (% from max)**
Maximum Merging	320KB (100%)
Constant Dist.	320KB (100%)
Exponential Dist.	$\sim$ 292KB (91.3%)
Linear Dist.	$\sim$ 298KB (93.3%)
Arctan Dist.	$\sim$ 293KB (91.6%)

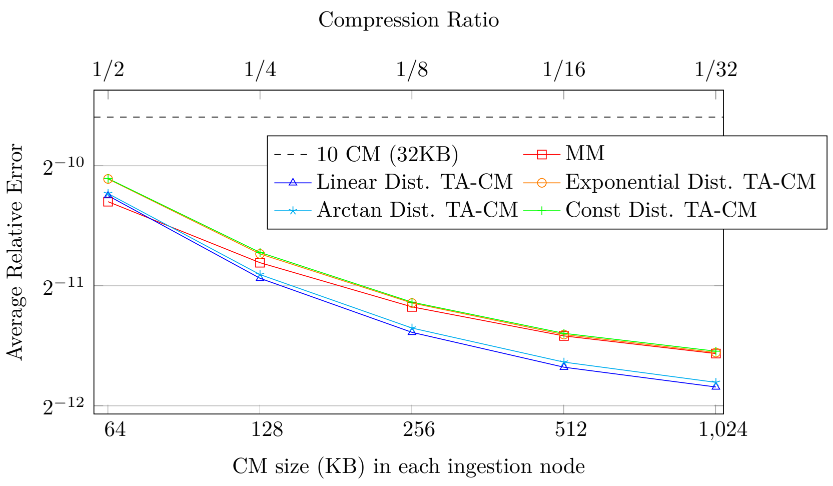


Figure 5.8: $n = 10$ nodes: Compression method comparison - various distributions

5.7.4 TA-KMV with n ingestion nodes

In this section, we evaluate *TA-KMV*. We use the same method as in the previous section to generate the input stream. However, for this section, we use only the linear distribution. We compare our *TA-KMV* with multiple compression ratios. The compression ratio is measured by *Total Hash-values Sent (THS)*. To the best of our knowledge, no compression scheme is available for this sketch, and therefore we compare our method only to the baseline, i.e., each ingestion node sends k hash values to the centralized server. Our measurement unit is the estimation precision rate to true cardinality. In this case, the number of hash values that are sent to the centralized server is $n \cdot k$, where n is the number of ingestion nodes.

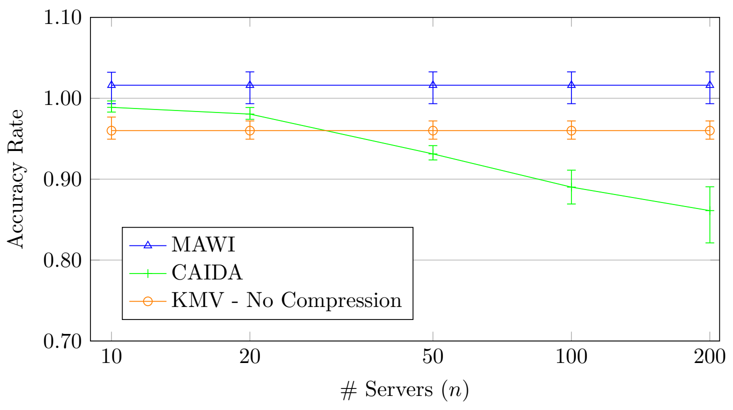


Figure 5.9: TA-KMV vs baseline for $k = 1024$.

Figure 5.9 depicts the impact of the number of ingestion nodes on the accuracy rate of TA-KMV, with the confidence interval. We define the accuracy rate as $\frac{\text{Estimation}}{\text{\# Flows}}$. The figure shows that the estimation remains fairly accurate even at 100 nodes. The CAIDA dataset does not have many flows, therefore the aggregate ends up receiving multiples of the same hash value, and thus the accuracy begins to drop. For example,

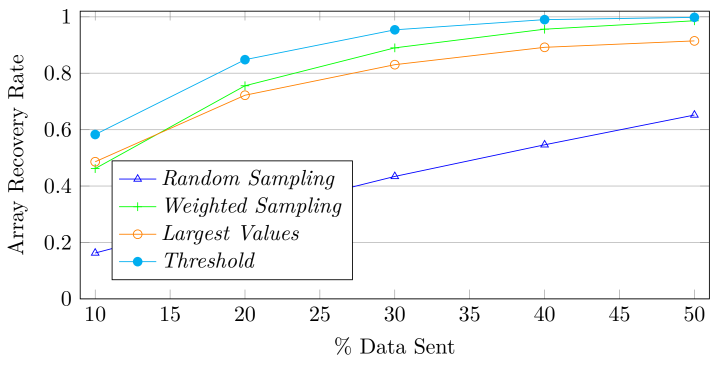


Figure 5.10: TA-HLL Array Recovery Rate in CAIDA – 10 ingestion nodes

at 200 nodes the central server receives only 300 different hash values. Contrast this with MAWI, where, due to the larger number of small flows, the central server receives all 1024 of the smallest hash values. Consider the case of 50 nodes. The baseline sends 51200 hash values, whereas the compressed version send around 4150 hash values – a 92% decrease for a near negligible decrease in accuracy. This creates a clear trade-off between the accuracy rate and the total summaries size. We note that sending less than $k \ln k$ hash values in total (e.g., k) greatly reduced the accuracy of the estimation.

5.7.5 TA-HLL with n ingestion nodes

Lastly, we compare the four methods for traffic aware cardinality estimation using compression of distributed HLL described in Section 5.6. Recall that the methods are:

(i) (Random) Each node reports each of its counters w.p. p.

(ii) (Weighted) Reporting a counter value c w.p. p_c.

(iii) (Largest) Each node reports its $\beta \cdot m$ largest counters.

(iv) (Threshold) Each node reports all its counters with values of at least a threshold c_T.

We evaluate the results of these four methods with an HLL array with size 128, using the CAIDA and MAWAI datasets. We evaluate all the four methods through two metrics for accuracy: The first metric is the centralized node Array Recovery Rate given as the percentage of cells in the centralized node recovered array that are identical to the corresponding cells in a HLL sketch that could be computed over all network traffic. The second evaluated metric is the relative error: For a distributed HLL estimation C, and real stream cardinality F, the relative error is $RE = \frac{|F-C|}{F}$. We first consider 10 ingestion nodes (and later vary their number).

Figure 5.10 shows the Array Recovery Rate for the four methods, as a function of percentage of data sent across the network as part of all ingestion nodes HLL array sizes. Here, with 10 ingestion nodes and HLL array size of 128, the total size is 1280. For instance, 20% means sending a total of 256 values to the centralized server. We can see that methods Weighted sampling (ii) and Threshold (iv) that require an iterative process with the centralized server achieve higher accuracy.

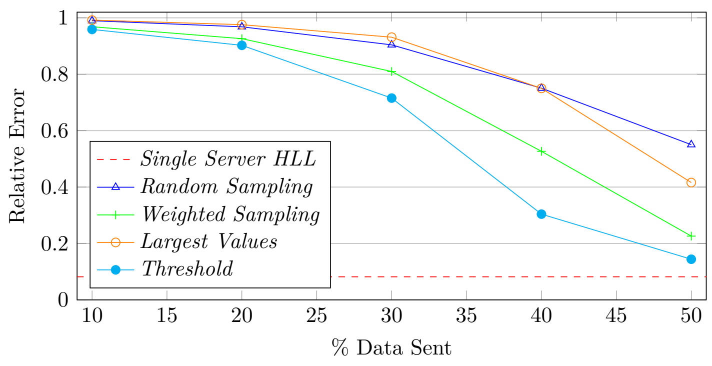


Figure 5.11: TA-HLL Relative Error in CAIDA – 10 ingestion nodes

Figure 5.11 presents the relative error of the same scenario, and as one could assume, the two graphs show coordinated results, i.e., the best method in terms of Array Recovery Rate (Threshold) allows lower Relative Error. Note that computing the particular threshold values for each node might require a relatively complicated iterative process with the centralized server with potential tradeoff between the number of iterations and communication overhead.

In Figures 5.12-5.14, we present the impact of the number of ingestion nodes across the network on accuracy. We used the same settings as described in previous figures,

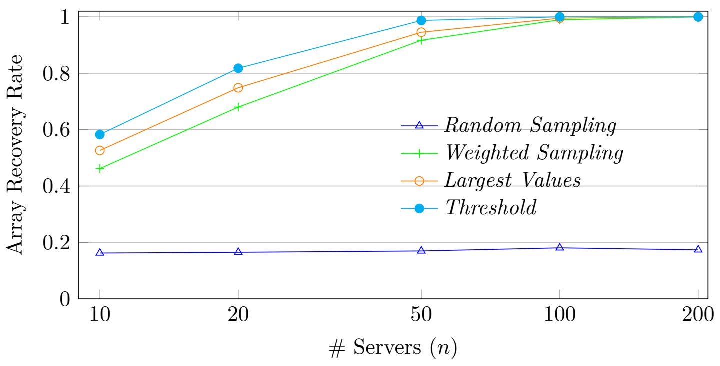


Figure 5.12: TA-HLL Array Recovery Rate in CAIDA – 10% data sent

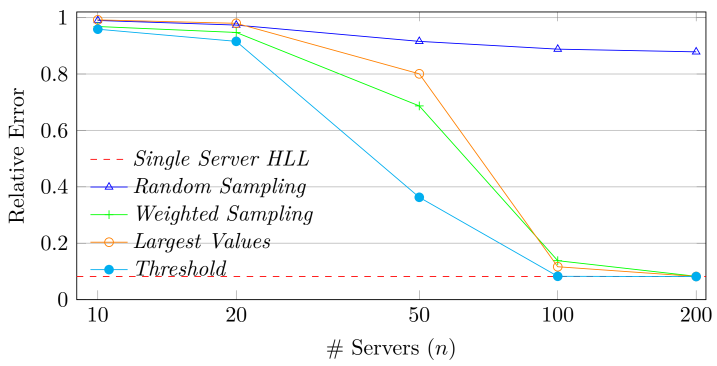


Figure 5.13: TA-HLL Relative Error in CAIDA - 10% data sent

with only 10% of the data sent to the centralized node and varied the number of ingestion nodes across the network. Once again the two methods with an iterative process with the centralized server are more accurate than the other two methods. Another observation is that as the number of ingestion nodes increases, the performance improves, that is because the total number of reports that arrive to the centralized node is increasing and as such, the probability to receive per each cell a value identical to that computed for the complete traffic is increasing as well.

5.8 Conclusion

In this paper, we presented the problem of merging data from multiple measurement points to one centralized server, described a distributed traffic-aware sketching scheme, and applied it to three unique sketches. We presented the CM-SKTC sketch as a simple, yet efficient method for compressing the CM sketch to any desired size, then used this method to generate *TA-CM*, a new scheme for flow-size measurements that provides high accuracy and decreases the total summaries size sent to the centralized server by considering the traffic of each node. This method is important in today's network design because when the traffic congestion in the network is high the need to successfully measure the network load is higher. In such situations, the extra load created by sending the summaries can worsen network congestion.

Moreover, we generalized this approach for cardinality estimation and introduced new traffic-aware designs of the KMV sketch as well as of the HLL sketch. They both send fewer values while retaining high accuracy cardinality estimation. Finally, we analyzed these sketches under multiple network settings and examined the trade-off between the accuracy and the size of summaries.

Several directions can be the focus in future work. A straightforward extension is developing compression algorithms for additional kinds of sketches allowing different measurement tasks (e.g., Quantiles for rank estimation [4]). Likewise, we wish to design further compression of sketches by leveraging existing generic compression techniques such as Huffman codes, LZ77 or gzip [65, 125, 36].

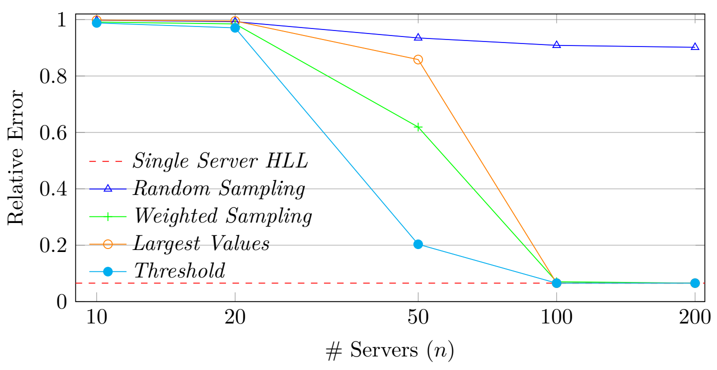


Figure 5.14: TA-HLL Relative Error in MAWI - 10% data sent

Chapter 6

Distributed Shared State Relaxed Sketches

In this chapter we present joint work appearing in [121]. We present an algorithm called *Strong Delayed-Writes (SDW)*, which enables distributed state management, and which guarantees consistent snapshots and r-relaxed strong linearizability facilitating implementation of distributed sketches.

The SDW protocol supports stream-order-agnostic mergeable data types like sketches. Variables of this type support three API functions: (1) UPDATE(v) – handling a single addition of element v, (2) QUERY() – returns a value based on the internal state, and MERGE(R') – merges the state of R' with that of the current variable. A requirement of any variable R fitting this model is that the QUERY result depends only on the set of elements that were ingested before it (either by an UPDATE or a MERGE), and not their order. We say that a query *reflects* an update, if the update altered the state before the query executed.

An execution of an algorithm renders a *history* H, which is a series of *invoke* and *response* events of the three API functions. In a *sequential history* each invocation is immediately followed by its response. The *sequential specification* $\mathcal{H}$ of a variable is its set of allowed sequential histories.

A *linearization* of a concurrent execution σ is a history $H \in \mathcal{H}$ such that after adding responses to some pending invocations and removing others, H and σ consist of the same invocations and responses and H preserves the order between non-overlapping operations [60]. If every concurrent execution has a linearization, we say that the variable is linearizable. For randomized variables we require a stronger property, called *strong linearizability.* The qualifier "strong" means that the linearization points are not determined post-facto, which is necessary in randomized variables [53].

A relaxed property of a variable is an extension of its sequential specification to allow for more behaviors. We adopt the notion of r-relaxed strong linearizability from Chapter 2, brought here for completeness. Intuitively, an r-relaxed variable allows a query to return a result based on all but at most r updates that happened before it.

Definition 6.0.1. 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 invocation that precede the same invocation in H'. The r-relaxation of $\mathcal{H}$ is the set of histories that have r-relaxations in $\mathcal{H}$, denoted $\mathcal{H}^r$.

Our SDW protocol is presented in Algorithm 9. To prove that Algorithm 9 is r-relaxed strongly linearizable, we first prove a helper lemma:

Lemma 6.0.2. *Consider a history H arising from a concurrent execution of Algorithm 9, and some completed update $u \in H$ executed by p_i. Let w be the value of win during u. Update u is reflected by every query q on any p_j, in every window $w' \geq w+2$.*

Proof. Let H be a history arising from a concurrent execution of Algorithm 9, and let $u \in H$ be some completed update executed by p_i. Let w be the value of *win* during the update's execution on p_i.

Update u is added to $objs[w \mod 3]$ on Line 12. On Line 23, $objs[(w+2) \mod 3]$ is broadcast to all switches, specifically to some switch p_j (as p_i retains the update in the same place that is merges received variables, this holds for $j = i$).

The next time p_j advances on Line 19, it enters window $w' = w+2$. Note that the variable that was queried in the previous window $(w'-1)$ is the same variable that reflected u. This variable is the one queried in round w', therefore reflected in round $w' = w+2$.

We now prove by induction that in round $w'' = w' + k$, u is reflected by a query in round w'' on p_j. The base is for $k = 0$, and has been prove.

Assume the hypothesis holds for $w' + l$, we prove for $w' + l + 1$. In round $w' + l$, u is reflected by $obj[(w' + l + 1) \mod 3]$. On Line 21, p_j merges this variable into $obj[((w' + l + 1) + 1) \mod 3]$, which is the variable queried in this round.

As this induction is true for all $k \geq 0$, it holds for any $w'' \geq w'$, proving the lemma. ☐

The following corollary follows directly from Lemma 6.0.2:

Corollary 6.1. *Let H be a history arising from a concurrent execution of Algorithm 9, and let $q \in H$ be some query completed by p_i. Let w be the value of win during its execution. Query q reflects all updates occurring in any window $w' \leq w - 2$.*

Note: A system where linearizability holds for sub histories including a single query is sometimes called *Ordered Sequential Consistency (OSC)* [79], this is commonly used in systems, e.g., ZooKeeper [66].

Finally, we define the *operation projection* of a history H and a set of operations O as the same history containing only invocations and responses of operations in O. We denote this $H|_O$ Using these formalisms we can prove the following theorem:

Theorem 6.2. *Consider a history H arising from a concurrent execution of Algorithm 9, and some query $q \in H$. Let U be the set of updates in H. The history of $H|_{U \cup \{q\}}$ is r-relaxed strongly linearizable.*

Proof. Let H be a history arising from a concurrent execution of Algorithm 9, let $q \in H$ be some query by p_i, and let U be the set of all updates in H. Denote $H|_{U \cup \{q\}}$ as H'. We show that H' is r-relaxed strongly linearizable with respect to $\mathcal{H}^r$, for $r = 2NB$. To prove this, we show the existing of two mappings, f and g, such that f maps operations in H' to *visibility points*, and g maps operations in H' to linearization points. Intuitively, visibility points are the time in the execution when an update is visible to a query, i.e., the query reflects the update. Bounding the number of preceding but not yet visible updates gives the relaxation.

We show that (1) $f(H') \in \mathcal{H}$, and (2) $g(H')$ is an r relaxation of $f(H')$. Together, this implies the theorem.

The visibility points ($f(H')$) are as follows:

- For the query, its visibility point is its return.
- For an update returning *false* at time t, its visibility point is t.
- For an update returning *true* at time t, let w be p_i's value of *win* at time t. The visibility point is the first time after t that p_i's value of *win* is *win* $+ 2$.

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

The linearization points ($g(H')$) are as follows:

- An update's linearization point is its return, either *true* or *false*.
- A query's linearization point is its return.

By definition, the linearization points as defined by $g(H')$ aren't decided post-facto – rather the linearization is a pre-determined point in the execution.

Consider some update $u \in H'$ executed on some p_j that returns *true*. Let w be p_j's value of *win* during its execution. Let w' be p_i's value of *win* during q's execution. We show that if $w \le w' - 2$, then q observes u, and if $w > w' - 2$, then q doesn't observe u.

From the definition of Algorithm 9, for any win_i on p_i and win_j on p_j, $|win_i - win_j| \le 1$.

If $w = w' - 2$, then when p_j added u to its local buffers, it did so to $obj[w \mod 3]$. As $|win_i - win_j| \le 1$, p_j advanced at least 1 window from w. When it did so, it sent $obj[w \mod 3]$ to p_i. In window $w' - 1$, p_i merges the update into $obj[w' + 1 \mod 3]$. In window w' this same variable is queried, thus q observes u. If $w \le w' - 3$, then the update is merged into some index of the variables array, and is copied over until it is reflected in all 3 of them, and specifically reflected in $obj[w' + 1 \mod 3]$ in window w'.

If $w \geq w' - 1$, then when p_j added u into its local buffer it did so to $obj[w \mod 3]$. This update is sent to p_i only in window $w+1$, and therefore isn't reflected in $obj[w'+1 \mod 3]$ in window w'.

Therefore, q reflects all updates that return true that happened during any window $w \leq w'-2$. As there are at most B updates that return true in any window, q reflects all but at most $2NB$ updates that precede it in H. Therefore, $g(H')$ is an $2NB$-relaxation of $f(H')$.

As the query returns a value based on the updates that happened before it, and each access to the process local state is down sequentially, q returns a value that reflects all successful updates that happen before it in $f(H')$. Therefore, $f(H') \in \mathcal{H}$. $\square$

Intuitively, every query returns a value reflecting a sub-stream of its preceding and concurrent updates, consisting of all but at most r successful ones. The upper bound r on the number of "missing" updates is of vast importance, without it the drift between one switch and another can grow in an unbounded fashion. For example, consider a counter distributed among two switches running an eventually synchronous algorithm. One switch can increment the counter an arbitrarily large number of times, while the other returns 0 on every query – the promise of eventual synchrony is too weak.

Algorithm 9 Algorithm running on switch p_i.

```
variables
    win, init 0
    $count$, init 0
    $objs$, init [$obj.init()$, $obj.init()$, $obj.init()$ ]
    buf, init {}
    rcvs, init {}
    acks, init {}

procedure UPDATE(v)
    if $count == B$ then                                          ▷ Write variable is full
        return false
    else
        $objs[win \mod 3].update(v)$                               ▷ Add to the write variable
        $count \leftarrow count + 1$ return true

procedure QUERY(arg)
    return $objs[win + 1 \mod 3].query(arg)$          ▷ Serve query from the read variable

procedure CHECK_DONE
    if $|rcvs| == n$ && $|acks| == n$ then
        $count \leftarrow 0$
        $win \leftarrow win + 1$                                          ▷ Rotate right
        $o' \leftarrow objs[win \mod 3]$
        $objs[win + 1 \mod 3].merge(o')$               ▷ Add the updates from window $w$
        $objs[win \mod 3] \leftarrow obj.init()$                          ▷ Clear write variable
        broadcast “$(objs[win + 2 \mod 3], win)$”                      ▷ Send sync message
        $rcvs \leftarrow \{i\}$
        $acks \leftarrow \{i\}$
        for all $(o', w')$ in $buf$ do                           ▷ Handle buffered messages
            $rcvs \leftarrow rcvs \cup \{j\}$
            $objs[win + 2 \mod 3].merge(o')$
            send “ack” to $p_j$
        $buf \leftarrow \{\}$

on receive “$(o', w')$” from $p_j$:                                               ▷ Sync
    if $w' > w$ then                                 ▷ Buffer messages from future windows
        $buf \leftarrow buf \cup \{(o', w')\}$
    else
        $rcvs \leftarrow rcvs \cup \{j\}$
        $objs[win + 2 \mod 3].merge(o')$                             ▷ Merge into sync buffer
        send “ack” to $p_j$
        return Check_Done()

on receive “ack” from $p_j$:
    $acks \leftarrow acks \cup \{j\}$
    return Check_Done()
```

Chapter 7

Conclusions and Future Work

Concurrent and distributed sketches are a fundemental tool in big data analysis, and more use-cases are being discovered constantly. In this thesis we have studied both concurrent and distributed sketches. We have presented both theoretical and practical work. We first presented a generic framework for parallelizing sketches and serving queries in real-time; the algorithm is strongly linearizable with regards to relaxed semantics. We discussed the necessity of adaptation for small streams, and how to implement such an adaptation. We showed the error bounds of two representative sketches, and implemented and evaluated the solution. We have shown it to be scalable and accurate when instantiated with the KMV sketch, and integrated it into the open-source Apache DataSketches library. We then identified two performance bottlenecks that arise in other sketches, namely the Quantiles sketch and the CountMin sketch, and analyzed them.

The first stemmed from the use of strong linearizabilty. While useful for inheriting the error analysis from the sequential setting, strong linearizabilty induces rigidity which may inhibit performance. We proposed IVL, a new correctness criterion relaxing linearizabilty. We showed that IVL has a number of desirable properties: First, like linearizability, it is a local property, allowing designers to reason about each part of the system separately. Second, also like linearizability but unlike other relaxations of it, IVL preserves the error bounds of PAC objects. Third, IVL is generically defined for all quantitative objects, and does not necessitate object-specific definitions. Finally, IVL is inherently amenable to cheaper implementations than linearizability in some cases.

The second performance bottleneck of our framework is due to the sequential merge operation. Some sketches have a longer merge operation, and it may grow longer as the stream grows larger. As update threads are blocked until the merger thread propagates their buffered elements, they are stalled until the merge finishes. We then briefly presented Quancurrent, a concurrent Quantiles sketch. We designed Quancurrent such that propagations are executed concurrently, thus providing a more scalable solution.

Finally, we studied sketches in the distributed setting. We presented the problem of merging data from multiple measurement points to one centralized server, described a distributed traffic-aware sketching scheme, and applied it to three unique sketches. We then analyzed these sketches under multiple network settings and examined the trade-off between the accuracy and the size of summaries. We have shown an algorithm for creating a distributed sketch, and presented its correctness proof.

We suggest multiple avenues for future work. Firstly, as shown in Chapter 4, there are limitations to the framework presented in Chapter 2. Whereas we can use the framework to parallelize any mergeable sketch, some sketches may benefit from tailored solutions, as we have shown for the Quantiles and CountMin sketches. It therefore remains to future work to identify additional sketches that do not fit within the framework, and parallelize them.

Secondly, we consider concurrent use-cases for Quancurrent. At its essence, given a stream drawn from a distribution, Quancurrent estimates the distributions CDF. Kraska et al. show that one can use an estimate for the CDF to construct an index structure, which they call a *learned index* [75]. They use machine learning tools to learn and estimate the CDF. Multiple other works have followed in their footsteps, enhancing and fine-tuning their structure [37, 74, 80, 46, 55, 113]. However, all these works still use ML tools to estimate the CDF. We believe that Quantiles sketches, and specifically Quancurrent, can be used to achieve comparable if not better throughput, with a lower memory footprint.

Finally, when distributing sketches we did not consider a failure model. In practical use-cases switches crash and nodes fail. To provide robust solutions for such failure model, we must design the distribute sketch to withstand such issues. To this end, we suggest analyzing the effects of crash-failures on the sketch's error, and the cost of recovering from a crash.

Bibliography

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

[2] Yehuda Afek, Anat Bremler-Barr, Shir Landau Feibish, and Liron Schiff. “Detecting heavy flows in the SDN match and action model”. In: *Computer Networks* 136 (2018), pp. 1–12.

[3] Yehuda Afek, Guy Korland, and Eitan Yanovsky. “Quasi-linearizability: Relaxed consistency for improved concurrency”. In: *International Conference on Principles of Distributed Systems.* Springer. 2010, pp. 395–410.

[4] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff M Phillips, Zhewei Wei, and Ke Yi. “Mergeable summaries”. In: *ACM Transactions on Database Systems (TODS)* 38.4 (2013), pp. 1–28.

[5] Vitalii Aksenov, Dan Alistarh, and Janne H Korhonen. “Scalable belief propagation via relaxed scheduling”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 22361–22372.

[6] 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, pp. 133–142.

[7] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. “The SprayList: A Scalable Relaxed Priority Queue”. In: *SIGPLAN Not.* 50.8 (Jan. 2015), pp. 11–20. URL: `http://doi.acm.org/10.1145/2858788.2688523`.

[8] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. “The spraylist: A scalable relaxed priority queue”. In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 2015, pp. 11–20.

[9] Amit Goyal, Hal Daumé III and Graham Cormode. “Sketch algorithms for estimating point queries in NLP”. In: *Joint conference on empirical methods in natural language processing and computational natural language learning.* 2012.

[10] Daniel Anderson, Pryce Bevan, Kevin Lang, Edo Liberty, Lee Rhodes, and Justin Thaler. “A high-performance algorithm for identifying frequent items in data streams”. In: *ACM Internet Measurement Conference*. 2017.

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

[12] Maya Arbel-Raviv and Trevor Brown. “Harnessing epoch-based reclamation for efficient range queries”. In: *ACM SIGPLAN Notices* 53.1 (2018), pp. 14–27.

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

[14] Athicha Muthitacharoen, Benjie Chen and David Mazières. “A low-bandwidth network file system”. In: *ACM symposium on Operating systems principles*. 2001.

[15] Michael D Atkinson, J-R Sack, Nicola Santoro, and Thomas Strothotte. “Min-max heaps and generalized priority queues”. In: *Communications of the ACM* 29.10 (1986), pp. 996–1000.

[16] Hagit Attiya, Faith Ellen, and Panagiota Fatourou. “The complexity of updating multi-writer snapshot objects”. In: *International Conference on Distributed Computing and Networking*. Springer. 2006, pp. 319–330.

[17] 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.

[18] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. “Models and issues in data stream systems”. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2002, pp. 1–16.

[19] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. “Counting distinct elements in a data stream”. In: *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer. 2002, pp. 1–10.

[20] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. “Understanding data center traffic characteristics”. In: *ACM SIGCOMM Computer Communication Review* 40.1 (2010), pp. 92–99.

[21] Hans-J. Boehm and Sarita V. Adve. “Foundations of the C++ Concurrency Memory Model”. In: *SIGPLAN Not.* 43.6 (June 2008), pp. 68–78. URL: `http://doi.acm.org/10.1145/1379022.1375591`.

[22] Arnon Boneh and Micha Hofri. “The coupon-collector problem revisited—a survey of engineering problems and computational methods”. In: *Stochastic Models* 13.1 (1997), pp. 39–66.

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

[24] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. "Unifying concurrent objects and distributed tasks: Interval-linearizability". In: *Journal of the ACM (JACM)* 65.6 (2018), pp. 1–42.

[25] Jacek Cichon and Wojciech Macyna. "Approximate counters for flash memory". In: *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*. Vol. 1. IEEE. 2011, pp. 185–189.

[26] 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. Snowbird, Utah, USA: ACM, 2014, pp. 88–99. URL: `http://doi.acm.org/10.1145/2594538.2594546`.

[27] Graham Cormode. "Data Sketching". In: *Queue* 15.2 (Apr. 2017), 60:49–60:67. URL: `http://doi.acm.org/10.1145/3084693.3104030`.

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

[29] Graham Cormode, Minos Garofalakis, Peter J Haas, and Chris Jermaine. "Synopses for massive data: Samples, histograms, wavelets, sketches". In: *Foundations and Trends in Databases* 4.1–3 (2012), pp. 1–294.

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

[31] Graham Cormode and S. Muthukrishnan. "An improved data stream summary: The Count-Min sketch and its applications". In: *J. Algorithms* 55.1 (2005), pp. 58–75.

[32] Graham Cormode, Shanmugavelayutham Muthukrishnan, and Ke Yi. "Algorithms for distributed functional monitoring". In: *ACM Transactions on Algorithms (TALG)* 7.2 (2011), pp. 1–20.

[33] Mayur Datar and Piotr Indyk. "Comparing Data Streams Using Hamming Norms". In: *Proceedings 2002 VLDB Conference: 28th International Conference on Very Large Databases (VLDB)*. Elsevier. 2002, p. 335.

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

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

[36] Peter Deutsch et al. *GZIP file format specification version 4.3*. Tech. rep. RFC 1952, May, 1996.

[37] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. "ALEX: an updatable adaptive learned index". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 969–984.

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

[39] Druid. *Druid*. `https://druid.apache.org/blog/2014/02/18/hyperloglog-optimizations-for-real-world-systems.html`. 2020.

[40] Marianne Durand and Philippe Flajolet. "Loglog Counting of Large Cardinalities". In: *European Symposium on Algorithms (ESA)*. 2003.

[41] Shaked Elias-Zada, Arik Rinberg, and Idit Keidar. *Quancurrent: A Concurrent Quantiles Sketch*. 2022. URL: `https://arxiv.org/abs/2208.09265`.

[42] Peter van Emde Boas, Robert Kaas, and Erik Zijlstra. "Design and implementation of an efficient priority queue". In: *Mathematical systems theory* 10.1 (1976), pp. 99–127.

[43] Cristian Estan and George Varghese. "Data streaming in computer networks". In: *Proceedings of Workshop on Management and Processing of Data Streams*. 2003.

[44] Cristian Estan and George Varghese. "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice". In: *ACM Transactions on Computer Systems (TOCS)* 21.3 (2003), pp. 270–313.

[45] William Feller. "An introduction to probability theory and its applications". In: *1957* ().

[46] Paolo Ferragina and Giorgio Vinciguerra. "The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds". In: *Proceedings of the VLDB Endowment* 13.8 (2020), pp. 1162–1175.

[47] Ivana Filipovi, Peter OHearn, Noam Rinetzky, and Hongseok Yang. "Abstraction for Concurrent Objects". In: *Theor. Comput. Sci.* 411.51–52 (Dec. 2010), pp. 4379–4398. URL: `https://doi.org/10.1016/j.tcs.2010.09.021`.

[48] Philippe Flajolet. "Approximate counting: a detailed analysis". In: *BIT Numerical Mathematics* 25.1 (1985), pp. 113–134.

[49] 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.

[50] Philippe Flajolet and G Nigel Martin. “Probabilistic counting”. In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*. IEEE. 1983, pp. 76–82.

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

[52] Frédéric Giroire. “Order statistics and estimating cardinalities of massive data sets”. In: *Discrete Applied Mathematics* 157.2 (2009), pp. 406–427.

[53] 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*. ACM. 2011, pp. 373–382.

[54] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. “PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs”. In: *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 2012, pp. 17–30.

[55] Ali Hadian and Thomas Heinis. “Considerations for handling updates in learned index structures”. In: *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 2019, pp. 1–4.

[56] Zijun Hang, Mei Wen, Yang Shi, and Chunyuan Zhang. “Interleaved Sketch: Toward Consistent Network Telemetry for Commodity Programmable Switches”. In: *IEEE Access* 7 (2019), pp. 146745–146758.

[57] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. “Network-wide heavy hitter detection with commodity switches”. In: *ACM Symposium on SDN Research (SOSR)*. 2018.

[58] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S Muthukrishnan, and Jennifer Rexford. “Carpe Elephants: Seize the Global Heavy Hitters”. In: *ACM Workshop on Secure Programmable Network Infrastructure*. 2020.

[59] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. “Quantitative relaxation of concurrent data structures”. In: *ACM SIGPLAN Notices*. Vol. 48. 1. ACM. 2013, pp. 317–328.

[60] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A Correctness Condition for Concurrent Objects”. In: *ACM Trans. Program. Lang. Syst.* 12.3 (July 1990), pp. 463–492. URL: `http://doi.acm.org/10.1145/78969.78972`.

[61] Stefan Heule, Marc Nunkesser, and Alexander Hall. “HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm”. In: *Proceedings of the 16th International Conference on Extending Database Technology*. 2013, pp. 683–692.

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

[63] Jaap-Henk Hoepman and John Tromp. “Binary snapshots”. In: *International Workshop on Distributed Algorithms*. Springer. 1993, pp. 18–25.

[64] Qun Huang, Xin Jin, Patrick PC Lee, Runhui Li, Lu Tang, Yi-Chao Chen, and Gong Zhang. “SketchVisor: Robust network measurement for software packet processing”. In: *ACM SIGCOMM*. 2017.

[65] David A Huffman. “A method for the construction of minimum-redundancy codes”. In: *IEEE Proc. of the IRE* 40.9 (1952), pp. 1098–1101.

[66] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. “ZooKeeper: Wait-free Coordination for Internet-scale Systems.” In: *USENIX annual technical conference*. Vol. 8. 9. 2010.

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

[68] Intel. *x86 Instruction Set Reference*. `https://c9x.me/x86/html/file_module_x86_id_327.html`. 2022.

[69] “Intel Tofino Series”. In: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[70] Amos Israeli and Asaf Shirazi. “The time complexity of updating snapshot memories”. In: *Information Processing Letters* 65.1 (1998), pp. 33–40.

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

[72] Nanxi Kang, Monia Ghobadi, John Reumann, Alexander Shraer, and Jennifer Rexford. “Efficient traffic splitting on commodity switches”. In: *ACM CoNEXT*. 2015.

[73] Z. Karnin, K. Lang, and E. Liberty. “Optimal Quantile Approximation in Streams”. In: *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. Oct. 2016, pp. 71–78.

[74] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. “RadixSpline: a single-pass learned index”. In: *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 2020, pp. 1–5.

[75] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. “The case for learned index structures”. In: *Proceedings of the 2018 international conference on management of data*. 2018, pp. 489–504.

[76] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. “Sketch-based change detection: Methods, evaluation, and applications”. In: *ACM Internet Measurement Conference*. 2003.

[77] Leslie Lamport. “Concurrent reading and writing of clocks”. In: *ACM Transactions on Computer Systems (TOCS)* 8.4 (1990), pp. 305–310.

[78] Leslie Lamport. “On interprocess communication”. In: *Distributed computing* 1.2 (1986), pp. 86–101.

[79] Kfir Lev-Ari, Edward Bortnikov, Idit Keidar, and Alexander Shraer. “Composing ordered sequential consistency”. In: *Information Processing Letters* 123 (2017), pp. 47–50.

[80] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. “LISA: A learned index structure for spatial data”. In: *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2020, pp. 2119–2133.

[81] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. “FlowRadar: A better NetFlow for data centers”. In: *USENIX NSDI*. 2016.

[82] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. “One sketch to rule them all: Rethinking network flow monitoring with univmon”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. 2016, pp. 101–114.

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

[84] Nihar R Mahapatra and Balakrishna Venkatrao. “The processor-memory bottleneck: problems and solutions”. In: *XRDS: Crossroads, The ACM Magazine for Students* 5.3es (1999), p. 2.

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

[86] Hagar Meir, Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Yonatan Gottesman, Idit Keidar, Eran Meir, Gali Sheffi, and Yoav Zuriel. “Oak: a scalable off-heap allocated key-value map”. In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2020, pp. 17–31.

[87] Meletis Margaritis, Christos Fidas, Nikolaos Avouris and Vassilis Komis. “A peer-to-peer architecture for synchronous collaboration over low-bandwidth networks”. In: *PCI*. 2003.

[88] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. “Efficient computation of frequent and top-k elements in data streams”. In: *International Conference on Database Theory*. Springer. 2005, pp. 398–412.

[89] Robert Morris. “Counting large numbers of events in small registers”. In: *Communications of the ACM* 21.10 (1978), pp. 840–842.

[90] Shanmugavelayutham Muthukrishnan et al. “Data streams: Algorithms and applications”. In: *Foundations and Trends® in Theoretical Computer Science* 1.2 (2005), pp. 117–236.

[91] Hun Namkung, Daehyeok Kim, Zaoxing Liu, Vyas Sekar, and Peter Steenkiste. “Telemetry retrieval inaccuracy in programmable switches: Analysis and recommendations”. In: *ACM SIGCOMM Symposium on SDN Research (SOSR)*. 2021.

[92] Gil Neiger. “Set-linearizability”. In: *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*. 1994, p. 396.

[93] Nikita Ivkin, Zhuolong Yu, Vladimir Braverman and Xin Jin. “Qpipe: Quantiles sketch fully in the data plane”. In: *ACM CoNext*. 2019.

[94] Sean Ovens and Philipp Woelfel. “Strongly linearizable implementations of snapshots and other types”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 197–206.

[95] Presto. *HyperLogLog in Presto: A significantly faster way to handle cardinality estimation*. `https://engineering.fb.com/data-infrastructure/hyperloglog/`. 2020.

[96] Pedro Reviriego, Pilin Junsangsri, Shanshan Liu, and Fabrizio Lombardi. “Error-Tolerant Data Sketches Using Approximate Nanoscale Memories and Voltage Scaling”. In: *IEEE Transactions on Nanotechnology* 21 (2022), pp. 16–22.

[97] Pedro Reviriego, Jorge Martínez, and Marco Ottavi. “Soft Error Tolerant Count Min Sketches”. In: *IEEE Transactions on Computers* 70.2 (2021), pp. 284–290.

[98] Hamza Rihani, Peter Sanders, and Roman Dementiev. “Multiqueues: Simpler, faster, and better relaxed concurrent priority queues”. In: *arXiv preprint arXiv:1411.1209* (2014).

[99] Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. “Fast concurrent data sketches”. In: vol. 9. 2. ACM New York, NY, 2022, pp. 1–35.

[100] Patrice Abry Romain Fontugne Pierre Borgnat and Kensuke Fukuda. “MAWILab: Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking”. In: *ACM CoNEXT*. 2010.

[101] Robert Rönngren and Rassul Ayani. “A comparative study of parallel and sequential priority queue algorithms”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 7.2 (1997), pp. 157–209.

[102] Ori Rottenstreich. “Sketches for Blockchains”. In: *International Conference on COMmunication Systems & NETworkS (COMSNETS)*. 2021.

[103] Ori Rottenstreich, Pedro Reviriego, Ely Porat, and S. Muthukrishnan. “Avoiding Flow Size Overestimation in the Count-Min Sketch with Bloom Filter Constructions”. In: *IEEE Transactions on Network and Service Management (TNSM)* 18.3 (2021), pp. 3662–3676.

[104] Yaniv Sadeh, Ori Rottenstreich, Arye Barkan, Yossi Kanizo, and Haim Kaplan. “Optimal Representations of a Traffic Distribution in Switch Memories”. In: *IEEE INFOCOM*. 2019.

[105] Christoph Scheurich and Michel Dubois. “Correct memory operation of cache-based multiprocessors”. In: *Proceedings of the 14th Annual International Symposium on Computer Architecture*. 1987, pp. 234–243.

[106] Anshumali Shrivastava, Arnd Christian Konig, and Mikhail Bilenko. “Time adaptive sketches (ada-sketches) for summarizing data streams”. In: *International Conference on Management of Data*. 2016.

[107] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. “Heavy-hitter detection entirely in the data plane”. In: *ACM Symposium on SDN Research (SOSR)*. 2017.

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

[109] “Source code.” In: https://github.com/dorh/Distributed-Sketch.

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

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

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

[113] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. “XIndex: a scalable learned index for multicore data storage”. In: *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2020, pp. 308–320.

[114] “The CAIDA Anonymized Internet Traces.” In: http://www.caida.org/data/overview.

[115] *Theta Sketch Equations*. `https://github.com/apache/datasketches-website/blob/master/docs/pdf/ThetaSketchEquations.pdf`. 2015.

[116] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. “Seedb: Efficient data-driven visualization recommendations to support visual analytics”. In: *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*. Vol. 8. 13. NIH Public Access. 2015, p. 2182.

[117] Jeffrey S Vitter. “Random sampling with a reservoir”. In: *ACM Transactions on Mathematical Software (TOMS)* 11.1 (1985), pp. 37–57.

[118] Yahoo! *Apache DataSketches*. `https://datasketches.apache.org/`. 2020.

[119] 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: *ACM SIGCOMM*. 2018.

[120] Minlan Yu, Lavanya Jose, and Rui Miao. “Software Defined Traffic Measurement with OpenSketch”. In: *USENIX NSDI*. 2013.

[121] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. “SwiSh: Distributed Shared State Abstractions for Programmable Switches”. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 171–191.

[122] Lior Zeno, Dan RK Ports, Jacob Nelson, and Mark Silberstein. “SwiShmem: Distributed shared state abstractions for programmable switches”. In: *ACM Workshop on Hot Topics in Networks (HotNets)*. 2020.

[123] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. “CocoSketch: high-performance sketch-based measurement over arbitrary partial key query”. In: *ACM SIGCOMM*. 2021.

[124] Zheng Zhang, Ming Zhang, Albert G Greenberg, Y Charlie Hu, Ratul Mahajan, and Blaine Christian. “Optimizing Cost and Performance in Online Service Provider Networks”. In: *USENIX NSDI*. 2010.

[125] Jacob Ziv and Abraham Lempel. “A universal algorithm for sequential data compression”. In: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.

DDoS ועוד. כדי לתמוך בניתוח בזמן אמת, בניית הסקיצה מבוצעת במשותף על ידי מספר צמתים, כל אחד צופה בחלק מהזרם, ומדווח מעת לעת על הסקיצה שלו לשרת מרוכז אחד שסוכם אותם. כדי למנוע עומס בתקשורת, כל צומת מדווח על גרסה דחוסה של הסקיצה שנאסף. באופן מסורתי, צמתים מדווחים באופן סימטרי על הסקיצות שלהם, כלומר דוחסים באותו יחס. אנו מסבירים שכדי למקסם את הדיוק של המדידה המשותפת, צמתים צריכים לדחוס את הסקיצה שלהם באופן שונה בהתבסס על כמות התעבורה שנצפה על ידי כל צומת. אנו ממחישים את הגישה לשלוש סקיצות נפוצות: סקיצת ה-Count-Min (CM) אשר מעריכה גדלי זרימה וכן עבור סקיצת ה-K-מינימום-ערכים (KMV) וה-HyperLogLog (HLL) ששניהם מעריכים את מספר הנבדלים. עבור כל סקיצה, אנו מחשבים יחסי דחיסה של צומת בהתבסס על התפלגות התעבורה. באופן כללי, זה נעשה בסבב אחד של תקשורת עם השרת המרכזי, שלאחריו ניתן לחשב את יחס הדחיסה עבור כל צומת. אנו מבצעים סימולציות נרחבות עבור הסקיצות ומראים בצורה אנליטית כי בתרחישים בעולם האמיתי, הסקיצות שלנו שולחות סיכומים קטנים יותר מאלה המסורתיים תוך שמירה על גבולות שגיאה דומים.

תקציר

עיבוד נתונים מהירים ובנפחים גבוהים מצריך לרוב ביצוע ניתוחים על ייצוג הנתונים, ולא על הנתונים עצמם. לשם כך, סקיצות נתונים הפכו לכלי הכרחי עבור חישוביםעיבוד נתונים מהירים ובנפחים גבוהים מצריך לרוב ביצוע ניתוחים על ייצוג הנתונים, ולא על הנתונים עצמם. לשם כך, סקיצות נתונים הפכו לכלי הכרחי עבור חישובים במהירות גבוהה על פני מערכי נתונים מסיביים. הם שומרים סיכום תמציתי של זרם הנתונים ועונים על שאילתות עליו (למשל, כמה אלמנטים ייחודיים יש, מהי תדירות האלמנטים) תוך שימוש בזיכרון מוגבל, במחיר של מתן קירובים ולא תשובות מדויקות. ספריות קיימות מספקות סקיצות עם אופטימיזציה גבוהה, אך אינן מאפשרות מקביליות ליצירת סקיצות באמצעות חוטים מרובים, או שאילתות בזמן בנייתן. בתזה זו אנו לומדים מספר היבטים של מערכות אלו.

תחילה אנו מציגים גישה גנרית למקבל סקיצות נתונים ביעילות ולאפשר שאילתות עליהם בזמן אמת, תוך הגבלת השגיאה שמיקבול כזו מוסיף. תוך שימוש בהרפיית הסמנטיקה וברעיון של ליניאריזציה חזקה, אנו מוכיחים את נכונות האלגוריתם שלנו ומנתחים את השגיאה של סקיצות ספציפיות. האלגוריתם שלנו משיג מדרגיות גבוהה תוך שמירה על השגיאה קטנה. תרמנו את אחת הסקיצות המקבילות שלנו לספריית סקיצות הנתונים בקוד פתוח. עם זאת, האלגוריתם שלנו מכילה שני צווארי בקבוק שבהם אנו מטפלים.

הראשון נובע מקריטריון הנכונות שנבחר. קריטריון הנכונות דה פקטו עבור אובייקטים מקבילים הוא יכולת ליניאריזציה. באופן אינטואיטיבי, תחת יכולת ליניאריזציה, כאשר קריאה חופפת עדכון, היא חייבת להחזיר את ערך האובייקט לפני העדכון או אחריו. לדוגמא, נבחן פעולת תוספת למונה הסופרת שלושה אירועים חדשים, ומקפיצה את הערך המונה מ-7 ל-10. בליניאריזציה של המונה, קריאה חופפת לעדכון זה חייבת להחזיר 7 או 10. אנו שמים לב שבמקרי שימוש טיפוסיים, כל ערך ביניים בין 7 ל-10 יהיה גם מקובל. כדי ללכוד את מידת החופש הנוספת הזו, אנו מציעים ליניאריזצית ערך ביניים (IVL), קריטריון נכונות חדש שמרפה את יכולת הלינאריזציה כדי לאפשר החזרת ערכי ביניים, למשל 8 בדוגמה למעלה. באופן גס, זה מאפשר לקריאה להחזיר כל ערך שמתוחם בין שני ערכי החזרה שהם חוקיים תחת ליניאריזציה. אנו מראים שיישומי IVL של סקיצות יורשים את השגיאה של מקבילם הסדרתי. אנו גם מראים שקריטריון זה מאפשר לעקוף את צוואר הבקבוק הראשון שלנו. צוואר הבקבוק השני נובע מכלליות האלגוריתם. מכיוון שהוא מספק דרך כללית למקבל סקיצות, חלק מהסקיצות עשויות להתייעל מפתרונות ספציפיים. לשם כך, אנו מציעים את Quancurrent, סקיצה Quantiles מקבילית. אנו מסבירים מדוע צוואר הבקבוק קיים באלגוריתם שלנו, ומראים בקצרה כיצד אנו מעצבים את Quancurrent כדי לספק פתרון יעיל יותר.

לבסוף, אנו סוקרים סקיצות נתונים בסביבה המבוזרת. מדידות רשת חשובות לזיהוי גודש, התקפות

המחקר בוצע בהנחייתה של פרופסור עידית קידר, בפקולטה להנדסת חשמל .

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי-עת במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Arik Rinberg, Alexander Spiegelman, Edward Bortnikov, Eshcar Hillel, Idit Keidar, Lee Rhodes, and Hadar Serviansky. “Fast concurrent data sketches”. In: *ACM Transactions on Parallel Computing* 9.2 (2022), pp. 1–35.

Arik Rinberg and Idit Keidar. “Intermediate Value Linearizability: A Quantitative Correctness Criterion”. In: *Journal of the ACM (Accepted Jan 2023).* 2023.

Dor Harris, Arik Rinberg, and Ori Rottenstreich. “Compressing Distributed Network Sketches with Traffic-Aware Summaries”. In: *IEEE Transactions on Network and Service Management* (2022).

סקיצות נתונים מקביליות: תאוריה ויישום

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

אריק רינברג

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

סקיצות נתונים מקביליות: תאוריה ויישום

אריק רינברג