Monitoring Application Response Time With Quantile Sketches

Monitoring Application Response Time with Quantile Sketches

William Daly

A Thesis in the Field of Information Technology

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

May 2019

Abstract

Monitoring systems track endpoint response times so human operators can detect and diagnose performance problems in web applications. One important kind of query answered by a monitoring system is called a quantile query. To answer quantile queries, most monitoring systems today either store an ever-growing dataset of response time measurements or pre-calculate summary statistics. This thesis considers whether mergeable quantile sketches can address limitations in current monitoring systems. By trading accuracy for space, sketches allow monitoring systems to answer quantile queries across arbitrary series and time windows while requiring less storage, reducing query response times, and increasing write throughput.

Acknowledgements

To my wife Zoe for her unrelenting love and support.

To my daughter Ada for taking naps and being wonderful.

To my thesis director Professor Jelani Nelson for guiding me through the strange and fascinating world of sketching algorithms.

Contents

List of Figures

List of Tables

List of Code

Chapter 1: Introduction

A search engine displays auto-complete suggestions on each keystroke. A streaming video service presents a personalized list of movie recommendations. An online auction closes just as hundreds of users submit bids. In each of these cases, speed is critical—users prefer services that respond quickly and reliably. The difference between responding in 50 milliseconds or 500 could mean the difference between user delight or frustration.

To users, web applications often appear simple. What could be easier than posting a 280-character tweet, viewing upcoming episodes of a TV show, or searching for hotel prices? In reality, modern web applications can consist of dozens or hundreds of network applications working together to respond to user requests. Typically, a service processing a user request initiates its own requests to specialized backend services, then combines the results into a response back to the user. Likewise, each of the backend services may initiate their own requests. As Dean & Barroso (2013) observe, in such a system *tail latency* becomes an important measure of the application's health. Slow responses, even if infrequent, can cumulatively delay the response to the user.

For this reason, operations teams responsible for web applications almost always monitor response times to detect and diagnose production issues. Often, an increase in the response time of an endpoint can indicate a user-facing problem. Since a web application can consist of many interacting services, each horizontally scaled across multiple machines and datacenters, response time data must be collected into

a central location where it can be queried efficiently. This is the responsibility of an *application monitoring system*: to make operational data, such as response times, available to human operators and automated alerting systems in a useful form.

The requirements of an application monitoring system pose several engineering challenges. First, such systems must handle high insert rates. For example, Facebook's Gorilla system was designed to handle over 700 million insert operations per minute (Pelkonen et al., 2015), and InfluxDB supports over 350,000 inserts per second on a single instance (Dix, 2016). Second, these systems must support queries of metrics data at interactive speeds for the detection and resolution of operational issues. Third, applications generate an enormous volume of operational data. Facebook's applications generate multiple terabytes of compressed data per day, with the data volume expected to double each year (Pelkonen et al., 2015); similarly, Gan et al. (2018) report that an application monitoring team at Microsoft collects billions of telemetry events daily. Application monitoring systems must therefore ingest and store an ever-increasing volume of operational data.

One particularly important capability of an application monitoring system is calculating *quantiles* for response time data. A quantile is a statistic that can summarize the tail of a distribution. For example, the 0.999-quantile of a dataset is the smallest value greater than or equal to 99.9% of the values in the dataset. When the values represent application response times, this is called the "99.9th percentile response time," and it is closely monitored by operations teams. For instance, both Google (Dean & Barroso, 2013) and Amazon (DeCandia et al., 2007) define the performance requirements of their services in terms of the 99.9th percentile response time. Unlike other summary statistics, such as the mean, the 99.9th percentile can detect problems experienced by a small subset of users, even if these manifest in only a small fraction of requests (DeCandia et al., 2007). Additionally, as Dean & Barroso

(2013) observe, in large-scale distributed systems even low-probability performance degradations can have an outsize impact on the system's overall performance. For these reasons, application monitoring systems need the ability to answer quantile queries.

Unfortunately, quantile queries can be expensive for current application monitoring systems to answer. Calculating exact quantiles requires access to every datapoint. Given the volume of data in the application monitoring case, this approach can impose high costs for transmitting, storing, and retrieving response time data. Other systems pre-calculate quantiles of interest, for example, the 99th percentile response time of a particular endpoint on a particular server over a particular time interval. While this approach reduces the amount of data stored, the system loses the ability to aggregate the data in certain ways. Such a system cannot, for instance, calculate the 99th percentile response time for *all* servers. Likewise, if the system stores 99th percentiles for 10-second windows, it loses the ability to calculate the 99th percentile for larger time intervals, such as hours or days. Such aggregations are critical to understanding the behavior of the system when detecting and diagnosing performance problems.

A class of data structures called *mergeable quantile sketches* can reduce the amount of data stored while preserving the ability to answer arbitrary quantile queries. A mergeable quantile sketch compresses a dataset into a format that can be used to answer quantile queries *approximately*. In addition, such sketches can be *merged* to answer aggregate queries. Although the query results are approximate, the sketching algorithms guarantee that the error is bounded. This is promising for application monitoring use cases because consumers of operational data generally care more about trends than exact results. For example, an operator responding to an outage in a production system would care that the 99th percentile response time in-

creased rapidly before the system crashed. The same operator most likely would not care that the 99th percentile was *exactly* 564 ms. Approximate results are acceptable as long as they are "close enough" to the true behavior of the system.

While many application monitoring systems have been developed, most of the ones used in the industry today store every datapoint. Examples include Facebook's Gorilla/Beringei (Pelkonen et al., 2015) (Teller, 2018), Graphite (Davis, 2011), InfluxDB (InfluxData Inc, 2018), OpenTSDB (The OpenTSDB Authors, 2010), and Uber's M3 (Uber Engineering, 2018). The statsd system developed at Etsy calculates quantiles for each time window, but these cannot be aggregated after-the-fact (Etsy, 2016). Several of the time-series databases surveyed in Jensen et al. (2017) are used for monitoring applications and sensor networks, but while some of these systems support approximate queries, none use mergeable quantile sketches.[1] Recently, Gan et al. (2018) investigated the use of a mergeable quantile sketch based on statistical moments for answering high-cardinality aggregation queries, including application monitoring systems. The system developed in this thesis takes a similar approach, but uses a sketch data structure with different performance trade-offs and stronger theoretical guarantees.

This thesis seeks to answer the question, "Could mergeable quantile sketches provide the basis of an application monitoring system?" Chapter 2 begins the investigation by describing the architecture of such a system, contrasting it with the architectures of other systems used in the industry today. Then, Chapter 3 evaluates the mergeable quantile sketch invented by Karnin et al. (2016), introducing several optimizations to improve the sketch's performance in an application monitoring context. Chapter 4 describes how the system can efficiently store and retrieve sketches

[1]Much of the work in this area has focused on using "model-based" approximations in sensor networks. Like mergeable quantile sketches, these can be constructed locally at each sensor, then aggregated after-the-fact to answer approximate queries. For a survey, see Sathe et al. (2013).

using a key-value store, and Chapter 5 explains how sketches can be used to answer user queries. Chapter 6 evaluates the performance of a prototype running in a cloud environment, and Chapter 7 suggests directions for future research.

Chapter 2: System Architecture

In this chapter, we consider how to design an application monitoring system based on mergeable quantile sketches. First, section 2.1 surveys the architectures of several systems deployed today. Section 2.2 then explores the properties of mergeable sketch algorithms and their architectural impact. Finally, section 2.3 introduces a new design using mergeable quantile sketches that will be tested at scale in Chapter 6.

2.1. Existing Systems

Imagine that developers have instrumented a web application to record the response time of each HTTP request. Imagine further that the web application has been deployed to multiple machines, with a load balancer distributing requests among these machines. We wish to monitor the response time data across all machines to detect and diagnose performance problems in the application.

First, we define some terms. A *datapoint* is a tuple of three attributes: a *timestamp* representing the time that a measurement was taken, a *value* representing the measurement itself (application response time), and a *series* string representing the kind of measurement. We refer to a set of datapoints within a given time interval as a *window*. For example, three requests to an application endpoint called "purchase" might generate the datapoints (1545450640, 3277, "purchase"), (1545450698, 2135, "purchase"), and (1545450772, 2691, "purchase"). These datapoints represent the fact that the first request (at Unix timestamp 1545450640) took 3277 milliseconds, the

second request took 2135 milliseconds, and the third request took 2691 milliseconds. We would consider these datapoints part of any time window that overlaps the three timestamps; one example would be the window starting at timestamp 1545450000 and ending at 1545460000.

In some cases, it is necessary to use multiple attributes to identify a measurement. For example, one might want to know which application generated the measurement or the datacenter in which the request was handled. Some systems support "tagging" measurements with arbitrary attributes; other systems encode multiple attributes in the series string itself. For example, "myapp.uswest.purchase" could represent that a measurement of the purchase endpoint originated from an application called "myapp" in the "uswest" datacenter. For consistency, we will follow the latter convention, encoding attributes directly in the series string.

With those definitions in hand, we now consider systems used in industry today. We divide these systems into two main categories: systems that store *individual datapoints* and systems that store *time window summaries*.

An example of a system that stores individual datapoints is Graphite (Davis, 2011). In the simplest possible configuration, applications send datapoints directly to Graphite, which then persists the datapoints to disk. Since Graphite stores individual datapoints, it can answer queries such as:

- What was the median value for the time series "myapp.uswest.purchase" from 12am to 4am last night?
- What was the 99th percentile of response times for the "purchase" endpoint across all datacenters in ten-second intervals in the last 24 hours?

Notice that the first query requires Graphite to calculate the median from a set of datapoints in a single time series, filtered by timestamp range (12am to 4am).

In contrast, the second query requires Graphite to combine datapoints from *multiple* time series, group them into ten-second windows, and calculate the 99th percentile for each window. At a high level, Graphite can answer these queries by retrieving a subset of the datapoints it stores (filtered by timestamp range or series), grouping them into one or more time windows, then calculating a quantile for each time window. Graphite's ability to answer these queries is, therefore, a direct consequence of the design decision to store individual datapoints.

Systems that store individual datapoints are widely used in the industry today. Examples include InfluxDB (Dix, 2017) (InfluxData Inc, 2018), KairosDB[1] (KairosDB Team, 2015), OpenTSDB (The OpenTSDB Authors, 2010), Facebook's Gorilla (Pelkonen et al., 2015), and Uber's M3 (Uber Engineering, 2018). One challenge with this approach is the amount of data that must be stored. High-throughput web applications routinely process hundreds of thousands of requests per second, and software-as-a-service deployments can consist of hundreds of interacting web applications. Therefore, persisting every datapoint can lead to prohibitively high storage costs.[2]

A second category of systems avoids storing individual datapoints, thereby reducing the size of the dataset. One example of such a system is Etsy's statsd (Malpas, 2011). Like the other systems we have discussed, statsd receives individual datapoints from applications, usually with each datapoint transmitted as a single UDP packet. Unlike the other systems, however, statsd does not persist every datapoint it receives. Instead, it holds datapoints in memory for a given period of time, called the "flush

[1]Internally, KairosDB uses Apache Cassandra (Lakshman & Malik, 2010) as its storage engine.

[2]The Gorilla system developed at Facebook uses a delta-of-delta compression scheme to reduce the cost of storing individual datapoints, reporting an average of 1.37 bytes per datapoint (Pelkonen et al., 2015). However, Gorilla's compression scheme depends on the timestamps in each series being spaced evenly (e.g. one datapoint per second), which is usually not the case for application response time data.

interval." Once the flush interval ends, statsd calculates a statistical summary for each time series using the datapoints it currently holds in memory. Each summary consists of a calculated count, mean, min, max, and several percentiles (usually the median, 95th, and 99th percentiles). Statsd then transmits the summaries to a back-end storage system.[3]

The statistical summaries sent by statsd are often much smaller than the original dataset size. Suppose, for example, that a high-traffic application processes $50,000$ requests per second, and statsd is configured with a flush interval of 10 seconds. Suppose that each datapoint can be represented using 64 bits (two 32-bit integers, one for the timestamp and one for the value). A system that stored individual datapoints would use about 4 MB to represent the 10-second time window. Alternatively, assuming that statsd stores fewer than 10 summary fields, then statsd could represent the same time window using only 640 bytes. This dramatic reduction in storage space has made statsd a popular choice for monitoring applications at scale.

However, the approach used by statsd comes at a cost. Unlike systems that store individual datapoints, systems that store statistical summaries cannot answer arbitrary quantile queries across time windows or series. To return to the running example from before, the system could calculate the median response time for the "purchase" endpoint in each individual datacenter for each 10-second time window, but it could *not* calculate the median response time *across* datacenters (multiple series) or for the interval from 12am to 4am (multiple time windows). By storing only calculated statistical summaries, the system loses the ability to answer arbitrary quantile queries.

[3]Statsd supports several backend storage systems, including Graphite and InfluxDB (Etsy, 2016). When using Graphite, each summary field (mean, min, max, etc) is stored as a separate time-series, with the values representing the calculated summary for a window rather than the values of the original datapoints.

A related problem occurs during downsampling. Downsampling is a feature supported by Graphite, InfluxDB, and other time series databases to merge time windows in order to reduce storage space. In most application monitoring scenarios, data loses value as it ages—it is usually more important to understand the behavior of an application as problems occur rather than after-the-fact. Therefore, it is sometimes acceptable to merge older time windows in order to save space. For example, suppose the system stores datapoints representing the *count* of requests per second, measured every second. One might configure the system to combine one-second windows older than 24 hours into ten-second windows. This can be achieved by adding the counts from each group of ten windows, then storing the sum as the count of the (merged) ten-second window. Summary statistics such as count, min, and max can be merged easily; percentiles, however, cannot be merged without losing information. Knowing the 95th percentile from several time windows is not enough to infer the 95th percentile for the *union* of the time windows—one would need the original datapoints as well. The statsd documentation recommends averaging percentiles (Etsy, 2016), but this provides no guaranteed error bounds on the result.

Let us return to the issue of querying across multiple time series. One might propose the following as a workaround to the limitations of statsd. Instead of sending a single datapoint to measure response time, have the application send multiple datapoints with different series names.[4] For example, an application might measure the response time for an invocation of the "purchase" endpoint, then send two datapoints to statsd: one for the series "myapp.uswest.purchase" and another for "myapp.purchase." Statsd would then calculate the summary for both the individual datacenter ("uswest") as well as across multiple datacenters. However, this workaround complicates application code and increases the load on statsd, especially

[4]Anecdotally, I have seen this workaround used in multiple production systems.

when the series name encodes many attributes. More importantly, this approach requires application developers to anticipate which aggregations will be needed to detect or diagnose a future production problem. In the best case, this leads to unnecessary development effort; in the worst case, it means that the necessary data may not be available to monitor application health effectively. This can make it more difficult to detect and respond to production issues early enough to prevent an outage. It can also prevent developers from understanding the cause of a production issue, a necessary step to prevent the issue from recurring.

This workaround points to an additional limitation with the statsd approach. Taking the workaround to its extreme, suppose that all applications send a datapoint for the series "allrequests" for every processed request. A user could then retrieve the calculated summaries of the "allrequests" series to find the 99th percentile across *all* requests. Notice that in order to calculate summary statistics for this series, all the datapoints must be processed by the same statsd instance. It is possible to deploy statsd as a single instance that receives all datapoints; however, as the number of datapoints grows, this instance can quickly become a scalability bottleneck. In particular, the number of inserted datapoints increases linearly with the number of requests to the system. Eventually, the statsd instance will not be able to process the datapoints it receives, resulting in either dropped data (if, for example, the operating system's UDP buffers become full) or an outage of the monitoring system.

A natural way to address the scalability bottleneck would be to horizontally scale statsd across multiple hosts. However, each distinct series must still be sent to the same (single) statsd instance—otherwise, no instance will have all the datapoints required to calculate an accurate statistical summary for the series. One can introduce a proxy server to route datapoints based on series name, distributing the load across multiple statsd instances (Wang, 2015). Alternatively, one could use an approach

similar to that of memcached (Nishtala et al., 2013), where applications use consistent hashing to map each series name to a particular statsd server, then write to the server directly. However, either approach can still lead to "hot spots" if the series mapped to an instance have a large number of inserts per second. In effect, calculation of statistical summaries for a particular series cannot be distributed across multiple instances.

We have identified a number of limitations with the statistical summaries used by statsd. Fundamentally, these limitations occur because percentile summaries from different time windows cannot be merged without distorting the results. Without the ability to merge windows, systems using statsd cannot answer arbitrary quantile queries across series or time intervals. For the same reason, such systems cannot perform downsampling. Finally, without the ability to merge summaries, statsd cannot distribute processing of a series across multiple instances, limiting the scalability of the system.

The next section presents a family of data structures called *mergeable quantile sketches* that may address the limitations of existing systems.

2.2. Mergeable Quantile Sketches

Suppose you are a developer troubleshooting a production issue. When you look at a dashboard showing the application's response time, you would probably care that the 95^{th} percentile spiked before an outage. However, you are unlikely to care that the value was *exactly* 53 milliseconds; approximations suffice as long as they are "close enough" to the actual values. A "quantile sketch" is a data structure that reduces space at the cost of accuracy, a trade-off that often makes sense for application monitoring systems.

Formally, we define an ϵ-approximate Φ-quantile as follows. Let x be an element in a dataset D, let $n = |D|$, and let $r(x)$ be the rank of x (the index of x in the sorted order of D). Then an ϵ-approximate Φ-quantile is any element in D such that $(\Phi - \epsilon)n \leq r(x) \leq (\Phi + \epsilon)n$. An ϵ-approximate quantile *sketch* is a data structure that supports ϵ-approximate quantile queries. In other words, the sketch returns an approximate result whose rank is within ϵ of the exact quantile. Most quantile sketches operate on streaming data, where the size of the input stream is not known in advance.[5] Such quantile sketches support two operations:

- $insert(S, x)$ adds an element x to the dataset maintained by the sketch S.
- $query(S, \Phi)$ returns an ϵ-approximate Φ-quantile for the dataset represented by the sketch S.

Wang et al. (2013) distinguish *deterministic* sketches from *randomized* sketches. Suppose someone performs a sequence of $insert(\cdot)$ operations followed by a $query(\cdot)$ on an empty sketch. Later, the person performs the same sequence of operations on another empty sketch. If the sketch is deterministic, then each query will return the same result. In contrast, if the sketch is randomized, then the query may return different results for different executions of the sequence. The reason is that a randomized sketch relies on coin flips to make decisions, whereas the deterministic sketch does not. Queries on randomized sketches are guaranteed to satisfy the error bound with constant probability $1 - \delta$. Although randomized sketches may violate the error bounds when $\delta > 0$, they usually require less space than deterministic sketches (Buragohain & Suri, 2009) (Wang et al., 2013).

Agarwal et al. (2013) introduced the concept of *mergeability* for quantile sketches. A mergeable sketch supports an additional operation $merge(S_1, S_2)$, which

[5]Most of the research on quantile sketches has focused on streaming data, for example Buragohain & Suri (2009), Wang et al. (2013), Greenwald & Khanna (2001), and Karnin et al. (2016).

accepts two sketches, S_1 and S_2, and produces a combined sketch S'. More precisely, let $S_\epsilon(D)$ be an ϵ-approximate sketch of the dataset D, where D is a multiset of values. Then $merge(S_\epsilon(D_1), S_\epsilon(D_2))$ produces a combined sketch $S' = S_\epsilon(D_1 \uplus D_2)$, where $\uplus$ is the disjoint union operator. Note that S' has the same error bound as $S_\epsilon(D_1)$ and $S_\epsilon(D_2)$. Additionally, if the sketch is randomized, the $merge(\cdot)$ operation must not increase the probability δ of violating the error bound. This means that one can build an ϵ-approximate sketch by merging several sketches together. The resulting sketch provides the same error guarantee as if we had inserted all the datapoints into a single sketch directly.

Mergeable quantile sketches suggest an alternative to the designs we considered in the previous section. As we saw, existing systems choose between storing individual datapoints or statistical summaries. Mergeable quantile sketches offer a middle ground between these two approaches. Like systems that store individual datapoints, sketches can be merged at query time to answer arbitrary quantile queries across time windows and series. Unlike these systems, however, quantile sketches do not require the system to store every datapoint. The query results are approximate, but the sketch data structure guarantees that the error is bounded. The next section describes the architecture of such a system in detail.

2.3. Proposed Architecture

How would one architect an application monitoring system based on mergeable quantile sketches? Each quantile sketch represents some set of datapoints, preserving enough information to answer quantile queries approximately. One could, therefore, use quantile sketches to represent windows of datapoints, similar to how statsd represents each window of datapoints using a statistical summary. Unlike statistical

summaries, however, sketches can be merged after-the-fact to answer arbitrary quantile queries with guaranteed error bounds.

For example, consider the sketches for the two series depicted in Figure 2.1. The backend stores multiple sketches, depicted as boxes. Each sketch corresponds to a series (either "us_east" or "us_west") and a time window (one of the ten-second intervals). To respond to a query for the 95$^{\text{th}}$ percentile response time across both datacenters from 00:10 up to 00:30, the backend would (1) retrieve sketches $A2$, $A3$, $B2$, and $B3$, then (2) merge the sketches into a new sketch M, then (3) calculate $query(M, 0.95)$, the approximate 95th percentile.

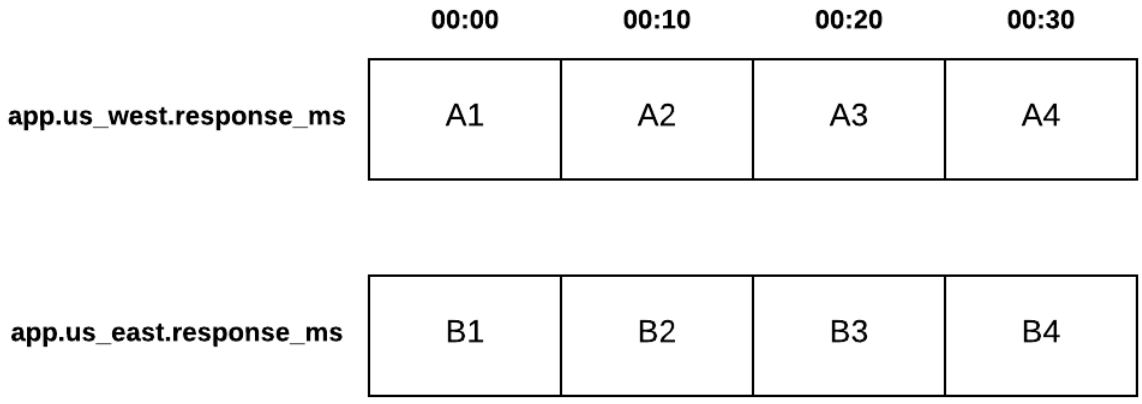

Figure 2.1: Quantile sketches representing time windows

Using mergeable sketches in this way avoids some limitations of existing systems. On the one hand, the system avoids storing every datapoint, reducing network usage and storage space. On the other hand, unlike the statistical summaries stored by statsd, quantile sketches can be merged at query time. Although the results are approximate, the sketch data structure guarantees that the error is bounded.

Figure 2.2 depicts a high-level architecture for such a system. First, the system must ingest response-time data from the application. The ingestion process involves

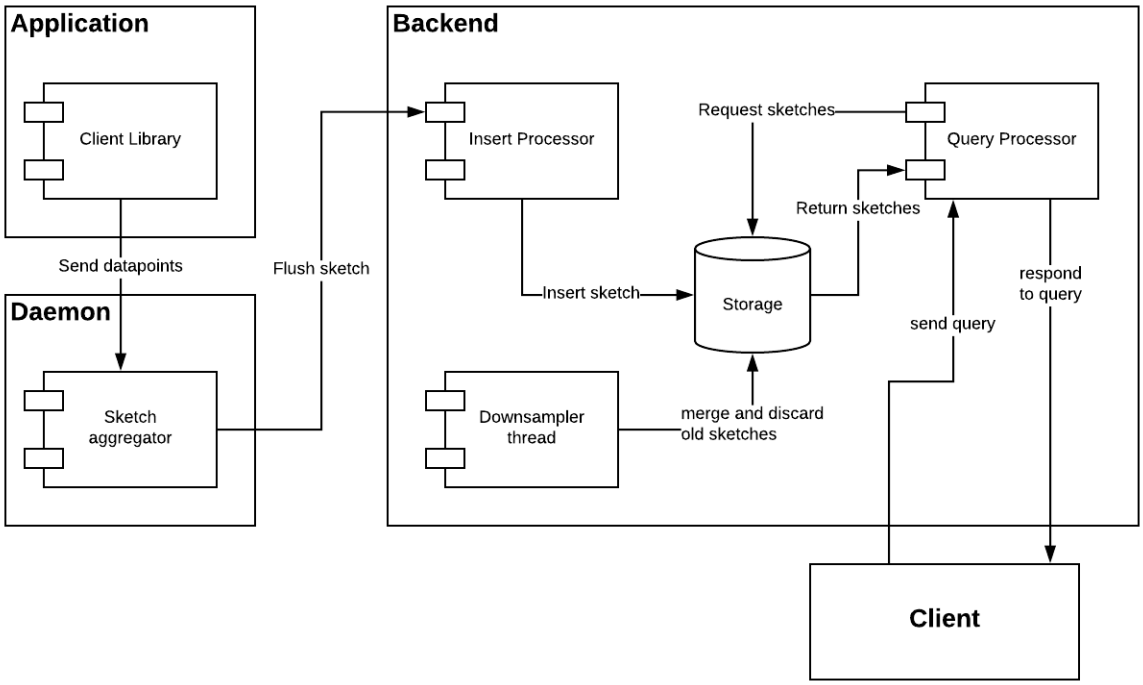

Figure 2.2: Proposed architecture based on mergeable quantile sketches

the following steps:

1. First, the application measures the response time for an operation, then sends the response time datapoint to a daemon process. As we discussed earlier, each datapoint consists of a series (string), timestamp (integer), and value (integer). The series identifies the operation being measured, and the value represents the response time of the operation.

2. For each distinct series, the daemon maintains a quantile sketch. When the daemon receives a datapoint via UDP, it checks if it has already initialized a sketch for the datapoint's series. If not, it will create an empty sketch for the series; otherwise, it retrieves the sketch already stored in memory. The daemon then inserts the datapoint into the sketch.

3. The daemon periodically flushes all sketches to the storage backend. Along with each sketch, the daemon sends the time window for the sketch and the series name. The time window consists of a start and end time, where the start time is the minimum timestamp of datapoints inserted into the sketch, and the end timestamp is the instant at which the window was flushed.

4. The storage backend writes the sketch to a persistent key-value store. The key is a composite of the sketch's series name and start timestamp, and the value is the sketch itself. If two sketches are received for the same key, the storage backend merges them.

Once the datapoints have been ingested, they become available for queries. Clients can query the backend service, which retrieves the relevant sketches from persistent storage. If necessary, the query processor merges sketches to answer queries across multiple time windows or series. Finally, the query processor performs a quantile query on each sketch and sends the results back to the client.

This architecture leverages the core operations of mergeable quantile sketches to address the limitations we saw in existing systems. First, the ability to merge sketches allows the system to answer arbitrary quantile queries across time windows or series with guaranteed error bounds and without storing the full dataset. Second, the system can downsample data by merging older sketches together to represent larger time windows, an operation that was not possible with the statistical summaries used by statsd. Chapter 5 will describe exactly how the system merges sketches at query time, and Chapter 4 will evaluate the space savings from downsampling by merging quantile sketches.

Finally, the ability to merge sketches avoids the scalability limitations we saw with statsd. We observed earlier that a system like statsd must process each series

on a single instance in order to calculate accurate statistical summaries. In contrast, the proposed architecture can distribute sketch calculation for the same series across multiple daemon instances. Each daemon generates a sketch for the series based on the datapoints it has received; these sketches can then be merged by the backend service on ingestion. Even if many application instances insert datapoints for the same series, the system can distribute processing of those datapoints across multiple daemon instances, thus avoiding the scalability bottleneck.

2.4. Conclusion

In this chapter, we analyzed the architecture of several application monitoring systems. We then considered a different architecture based on quantile sketches. Unlike existing systems, this architecture avoids the need to store the full dataset while retaining the ability to answer quantile queries across multiple time windows and series.

However, it remains to be seen whether such a system is feasible. We begin to answer this question in the next chapter, where we consider a specific mergeable quantile sketch implementation. To provide the basis of a feasible application monitoring system, the implementation must significantly reduce the size of the dataset, introduce minimal error during insert/merge operations, and perform inserts/merges quickly. As we will see, the sketch algorithm invented by Karnin et al. (2016) is a plausible candidate.

Chapter 3: Quantile Sketch Implementation

In this chapter, we consider the details of a mergeable quantile sketch implementation suitable for an application monitoring system, focusing on the sketch invented by Karnin et al. (2016). Along the way, I identify several optimizations to the KLL compaction algorithm that are important for the application monitoring use-case. This optimized implementation is then evaluated experimentally on a series of micro-benchmarks.

3.1. KLL Sketch

Karnin et al. (2016) developed a randomized, mergeable quantile sketch called KLL.[1] The authors present several variations of the KLL sketch; we focus here on the version with random sampling, but not with GK sketches.[2]

The KLL sketch consists of a *sampler* and one or more *compactors*, as shown in Figure 3.1. Initially, the sketch has only one compactor and the sampler preserves every item (no sampling). Incoming items from the data stream are inserted into the first compactor. When a compactor reaches its capacity, it sorts its stored items, randomly discards either even- or odd-indexed items, then outputs the surviving items to the next compactor in the hierarchy, adding a new compactor if necessary. As more compactors are created, compactors lower in the hierarchy shrink their capacity. In

[1] The sketch is very similar to the one presented in Agarwal et al. (2013), except that KLL sets the compactor capacities differently and uses random sampling to achieve lower space complexity.

[2] Using GK sketches for the top compactors potentially gives up mergeability, which we need for the application monitoring system.

particular, compactor capacity decreases exponentially as one descends the hierarchy, except for the top $O\left(\log \log \frac{1}{\delta}\right)$ compactors, which have a fixed size k.

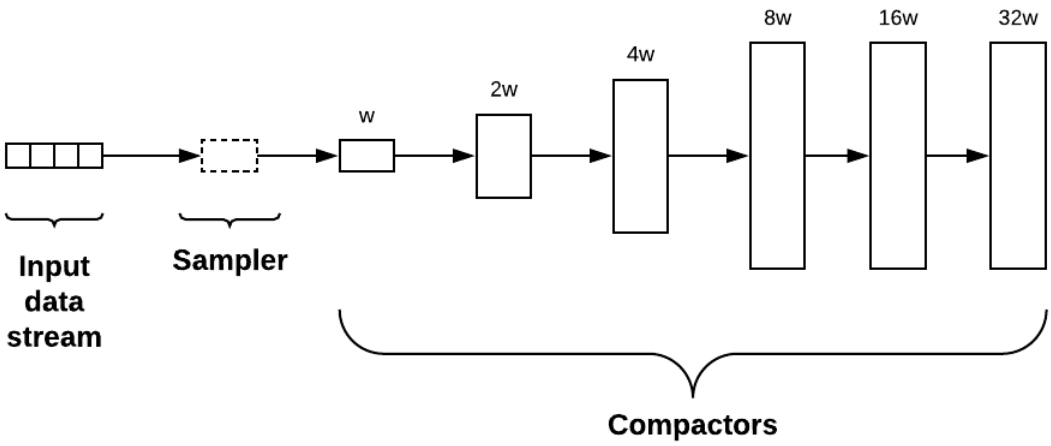

Figure 3.1: A KLL sketch consists of a sampler and one or more compactors. The height of each rectangle represents the compactor's capacity. Multiples of w represent the weight of the items in each compactor.

The minimum compactor capacity is 2, since each compactor must have at least one item to discard and one item to preserve during compaction. In effect, these compactors randomly sample one out of every two items received. It follows that a sequence of c compactors with capacity 2 is equivalent to sampling one out of every 2^c items. The sketch's sampler simulates these compactors in $O(1)$ space using a weighted version of the reservoir sampling algorithm presented in Vitter (1985), thus reducing the sketch's worst-case space complexity.

Each item stored in the sketch represents some number of items in the original

data stream. This number is referred to as the *weight* of the item. When a compactor discards half of its items, the weight of the surviving items doubles, since each surviving item now represents twice as many items from the original data stream. This allows the sketch to represent the original data stream accurately without storing every item. All of the items in a compactor have the same weight, and this weight is determined by the compactor's level in the hierarchy.

One can query a KLL sketch using the approach suggested in Wang et al. (2013). First, the items from all compactors are pre-processed to calculate an "estimated rank" for each item:

1. Sort the items from all compactors by value.
2. Iterate through the items in sorted order, maintaining a running sum of the item weights. Store the running sum at each index as the "estimated rank" for the item at that index.

One can then calculate an ϵ-approximate Φ-quantile by performing a binary search to find the item with the target rank closest to Φn, where n is the number of items from the input data stream. The time complexity of the algorithm is dominated by the sort operation in the pre-processing step.[3]

The merge operation works by concatenating the items from the compactors from both sketches at each level in the hierarchy. The sketch can then run compaction on sketches that exceed their capacity in order to reduce the number of items. While the sampler preserves all incoming items, this procedure is sufficient

[3]It is possible to reduce the time complexity to $O(n)$ on average using a variation of the algorithm presented in Hoare (1961). One would need to augment the algorithm to work over weighted items. As an optimization, one could also recast queries for $\Phi > 0.5$ to an equivalent query for $(1 - \Phi)$ over items sorted in reverse order, ensuring that the algorithm partially sorts at most half the items. This approach can be more efficient when performing a single quantile query for a sketch; however, when performing multiple quantile queries, it is likely faster to amortize the pre-processing cost over multiple binary searches.

to produce a mergeable summary. The process becomes more complicated, however, when a sketch's sampler has absorbed at least one compactor. Let r_1 and r_2 denote the sample rates of the two sketches to be merged, and let s denote the sampler with the greater sample rate, breaking ties arbitrarily. Then any compactor with weight less than $\max(r_1, r_2)$ must output its weighted items to s. Likewise, the other sampler must output its weighted item (if any) to s. As Karnin et al. (2016) prove, the resulting sketch provides the same space and error guarantees as an equivalent sketch generated from the original input data.

Karnin et al. (2016) show that the sketch can answer ϵ-approximate Φ-quantile queries with probability $1-\delta$ using $O\left(\frac{1}{\epsilon}\log^2\log\frac{1}{\delta}\right)$ space. To the best of my knowledge, this result gives the lowest space complexity yet achieved.

3.2. Alternatives Considered

This section surveys alternatives to the KLL sketch that have been proposed in the literature, focusing on four sketches that are known to be mergeable: Count-Min (Cormode & Muthukrishnan, 2005), Q-Digest (Shrivastava et al., 2004), MRL99 (Manku et al., 1999), and M-Sketch (Gan et al., 2018). The discussion excludes sketches that are not known to be mergeable summaries, as defined by Agarwal et al. (2013); this includes GK (Greenwald & Khanna, 2001), FO (Felber & Ostrovsky, 2015), and t-digest (Dunning & Ertl, 2019). Table 3.1 summarizes the space and time complexity for each sketch surveyed.

The Count-Min sketch can answer ϵ-approximate quantile queries over a fixed universe of u values. Unlike the KLL sketch, the Count-Min sketch supports deletions. However, this comes at the cost of additional space complexity: to answer ϵ-approximate quantile queries, the Count-Min sketch requires $O\left(\left(\frac{1}{\epsilon}\right)\log^2 u \log\frac{\log u}{\delta}\right)$

Sketch	**Space**	**Insert Time**	**Query Time**[1]
Count-Min[2]	$O\big(\big(\frac{1}{\epsilon}\big)\log^2 u \log\frac{\log u}{\delta}\big)$	$O\big(\log u \log\frac{\log u}{\delta}\big)$	$O\big(\log u \log\frac{\log u}{\delta}\big)$
Q-Digest[3]	$O\big(\frac{1}{\epsilon}\log u\big)$	$O\big(\log\frac{1}{\epsilon} + \log\log u\big)$	$O(\lvert S\rvert \log \lvert S\rvert)$
MRL99[4]	$O\big(\frac{1}{\epsilon}\log^2\frac{1}{\epsilon}\big)$	$O\big(\log\frac{1}{\epsilon}\big)$	$O(\lvert S\rvert \log \lvert S\rvert)$
M-Sketch[5]	$O(k)$	$O(k)$	$O(k^2)$ per iteration of Newton's method
KLL[6]	$O\big(\frac{1}{\epsilon}\log^2\log\frac{1}{\delta}\big)$	$O\big(\log\frac{1}{\epsilon} + \log\log\log\frac{1}{\delta}\big)$	$O(\lvert S\rvert \log \lvert S\rvert)$

[1] The size of the sketch is denoted by $|S|$.
[2] These bounds are proven in Cormode (2009).
[3] The space complexity is given in Buragohain & Suri (2009). The (amortized) insert time complexity is for the optimized version of Q-Digest presented in Wang et al. (2013). The query time is dominated by sorting the nodes of the Q-Digest.
[4] The space bound is proven in Manku et al. (1999), and the (amortized) insert time complexity is given in Wang et al. (2013). The query time is dominated by sorting the retained values.
[5] The parameter k is the number of moments stored in the sketch.
[6] The space bound is proven in Karnin et al. (2016). The query time is dominated by sorting the retained values. See Appendix A for a proof of the insert operation time complexity.

Table 3.1: Summary of space and time complexity for surveyed sketches

space (Cormode & Muthukrishnan, 2005).

Q-Digest is another fixed-universe sketch with space complexity $O\left(\frac{1}{\epsilon} \log u\right)$ (Buragohain & Suri, 2009). Compared to Count-Min, Q-Digest cannot support deletions, but it has better space complexity. Unlike KLL and Count-Min, which provide probabilistic error guarantees, Q-Digest is deterministic. However, for the application monitoring use case, probabilistic error guarantees are acceptable as long as the probability of failure is sufficiently low.

The space complexity of both Count-Min and Q-Digest depend on u, the size of the universe of possible input values. In the application monitoring case, the input values measure the durations of arbitrary operations at high precision (often milliseconds or nanoseconds). Therefore, the universe of possible input values can be large—nanosecond durations are typically represented using 32- or 64-bit integers. This suggests that sketches whose size depends on u might be sub-optimal for the application monitoring case.

Unlike Count-Min and Q-Digest, the MRL99 sketch presented in Manku et al. (1999) does not require a fixed universe of input values. As a result, its space complexity $O\left(\frac{1}{\epsilon} \log^2 \frac{1}{\epsilon}\right)$ depends only on the error bound ϵ. Like KLL, MRL99 unevenly samples the input stream by randomly discarding the even- or odd-indexed values stored in the sketch's buffers. Wang et al. (2013) show experimentally that MRL99 can insert values faster than Q-Digest because MRL99 interacts well with the CPU caches. As we will see later, KLL has similar performance characteristics.

During the writing of this thesis, Gan et al. (2018) developed a quantile sketch for efficient aggregation across large datasets. One of their target use cases is application monitoring. As we saw in Chapter 2, a key challenge in the application monitoring case is efficiently aggregating data across time windows and series. Gan et al. (2018) observe that when the number of merges exceeds about $10,000$, then the

cost of merging sketches dominates the time to answer user queries. For this reason, they develop a sketch, called a "moments-sketch" or "M-Sketch," that can be merged in under 50 ns.

The M-Sketch consists of a counter, the min and max values of the dataset, and k sample moments.[4] Both the insert and merge operations can be implemented in $O(k)$ time. However, the query operation is fairly expensive. The sketch must solve a convex optimization problem to find a probability distribution that has the same moments as the sampled moments in the sketch while maximizing entropy. Gan et al. (2018) conclude that the M-Sketch can answer user queries faster when the number of merges is at least 10^4; if the number of merges is less than 100, however, the query time dominates and the M-Sketch performs worse than the other sketches. Unlike the KLL sketch, the space used by M-Sketch depends on the distribution of the values in the input data stream. Additionally, some inputs and values of k can cause numerical instability, negatively impacting query performance and accuracy (Gan et al., 2018).

We now turn to the implementation details of KLL sketches. First, section 3.3 presents optimizations to compaction that significantly improve the performance of the KLL sketch. Section 3.4 then empirically evaluates the size, error, and execution time of the optimized KLL sketch implementation for different values of k.

3.3. Compaction Optimizations

To understand how KLL performs in practice, several benchmarks were executed on an implementation written in the Rust programming language. The code was then profiled using the Linux "perf" tool to identify bottlenecks in several bench-

[4]The sketch can optionally store log-moments as well to more accurately approximate long-tailed datasets.

marks.[5]. In this section, we will focus on the results from three benchmarks:

- **Insert**: Initialize a sketch with 4096 values, then measure the time to insert 2048 additional values.
- **Merge (sequential data)**: Initialize two sketches using the same sequence of 4096 values. Measure the time to merge the two sketches.
- **Merge (random data)**: Initialize two sketches using separate sets of 4096 randomly-generated values. Measure the time to merge the two sketches.

Listing 3.1: Naive implementation of KLL sketch compaction

```
// On input, overflow is empty
// On output, overflow is sorted (asc by value)
pub fn compact(&mut self, overflow: &mut Vec<u32>) {
    debug_assert!(overflow.is_empty());
    self.data.sort_unstable();

    let leftover = if self.data.len() % 2 != 0 {
        self.data.pop()
    } else {
        None
    };

    let n = self.data.len();
    let mut idx = rand::random::<bool>() as usize;
    while idx < n {
        overflow.push(self.data[idx]);
        idx += 2;
    }
    self.data.clear();

    if let Some(v) = leftover {
        self.data.push(v);
    }
}
```

[5]The benchmarks were executed on an 8-core, 3GHz AMD Ryzen machine with 32GB memory. The Rust "bencher" package was used to execute the benchmarks repeatedly until the timings converged to a stable median (The Rust Project Developers, 2018) For all benchmarks, the capacity of the top-level compactor was set to $k = 200$

The most direct way to implement compaction is shown in Listing 3.1. This implementation closely follows the algorithm as described in Karnin et al. (2016). First, the items in the compactor are sorted, then either the even- or odd-indexed items are discarded (if there are an odd number of items, the last item remains in the compactor); the rest are output to the next compactor. Profiling this implementation reveals that the insert benchmark spends over half its execution time sorting values in the compactor, and the merge benchmarks spend over 75% of their execution time sorting (Table 3.2).

Benchmark	% **Time Spent Sorting**
insert	57.89
merge (sequential data)	78.41
merge (random data)	77.25

Table 3.2: Time spent sorting in an unoptimized KLL sketch implementation

The profiling results suggest a simple optimization: sort the values once, then merge the sorted values during compaction. Initially, values are inserted into the first compactor in the order they arrive from the input data stream. Once the first compactor reaches its capacity, the values are sorted and half are discarded. The surviving values will already be in sorted order. Assuming that the values in the next compactor are also in sorted order, one can simply merge the two sorted sequences, maintaining the invariant that, except for the first one, compactors store values in sorted order.

A simple way to merge two sorted sequences is shown in Listing 3.2. At each iteration, the algorithm compares the first values in each sequence and selects the smaller of the two values. The index for that array is incremented, and the loop continues until all the values from one of the arrays has been consumed. Any remaining values are then copied to the output array.

Listing 3.2: Function that merges two sorted sequences, with conditional branches

```
1  fn merge_sorted(v1: &[u32], v2: &[u32]) -> Vec<u32> {
2      let (mut i, mut j) = (0, 0);
3      let (n, m) = (v1.len(), v2.len());
4      let mut result = Vec::with_capacity(n + m);
5      while i < n && j < m {
6          let (x, y) = (v1[i], v2[j]);
7          if x < y {
8              result.push(x);
9              i += 1;
10         } else {
11             result.push(y);
12             j += 1;
13         }
14     }
15     result.extend_from_slice(&v1[i..n]);
16     result.extend_from_slice(&v2[j..m]);
17     result
18 }
```

With fewer sort operations on the critical path, a new bottleneck emerges. For each of the $\frac{n}{2}$ iterations of the loop, line 16 of Listing 3.1 writes the selected value to the output vector. This can be inefficient, since the output vector may need to allocate additional space. On every iteration of the loop, the Rust vector implementation must check if the vector's size has exceeded its capacity; if so, it must copy every stored value to a new, larger array. Since it has no information about the total number of values that will be inserted, the vector may not allocate all the necessary memory upfront. Instead, it may need to grow its capacity multiple times to accommodate additional values.

One can avoid the cost of growing the output vector by bulk-copying the selected values, as shown in Listing 3.3. First, the algorithm moves the surviving values into the first $\frac{n}{2}$ positions of the compactor's data vector, overwriting the discarded values (lines 14-17). These values are then bulk-copied to the output buffer (line 19). This allows the Rust vector implementation to allocate enough space for the surviving values upfront. The values can then be copied into the output vector without

additional checks.[6] The result is a further 8% improvement on the insert benchmark, 22% on the merge sequential data benchmark, and 12% on the merge random data benchmark.

Listing 3.3: Compaction implementation that bulk-copies values to the output vector

```
pub fn compact(&mut self, overflow: &mut Vec<u32>) {
    debug_assert!(overflow.is_empty());
    self.ensure_sorted();  // sort ‘self.data‘ if it is not already sorted

    let n = self.data.len();

    let leftover = if n % 2 != 0 {
        Some(self.data[n - 1])
    } else {
        None
    };

    let mut idx = rand::random::<bool>() as usize;
    while idx < n {
        self.data[idx / 2] = self.data[idx];
        idx += 2;
    }

    overflow.extend_from_slice(&self.data[..n / 2]);

    self.data.clear();
    if let Some(v) = leftover {
        self.data.push(v);
    }
}
```

Unfortunately, the simple merge-sorted algorithm performs a comparison on every iteration of the loop. This means that a branch taken on any particular iteration depends on the input data. Modern CPUs try to predict which branch to take based on prior iterations, speculatively executing the expected branch. If the CPU mispredicts the branch, then the results of the speculative execution must be discarded. This is referred to as a “pipeline stall,” and it is an expensive operation.

[6] Another way to avoid avoid unnecessary reallocations would be to reserve $\frac{n}{2}$ capacity from the overflow buffer upfront, then push values directly to the overflow buffer. However, the Rust vector's push operation still checks if it needs to reallocate, introducing some additional overhead. As of this writing, the approach in Listing 3.3 performs slightly, but measurably, better.

We can see the effect of pipeline stalls in Figure 3.2 by comparing the two merge benchmarks. In the sequential version of the benchmark, both sketches are initialized with the same stream of (sequential) input values. When merging the sketches, the values in one sketch are always equal to the values in the other sketch, so the same branch is taken on each iteration of the loop. With this input data, the CPU can accurately predict the branch, so speculative execution improves performance. In contrast, when merging random data, different branches may be taken on different iterations of the loop. This negatively impacts branch prediction, thus increasing the benchmark execution time by 92%.

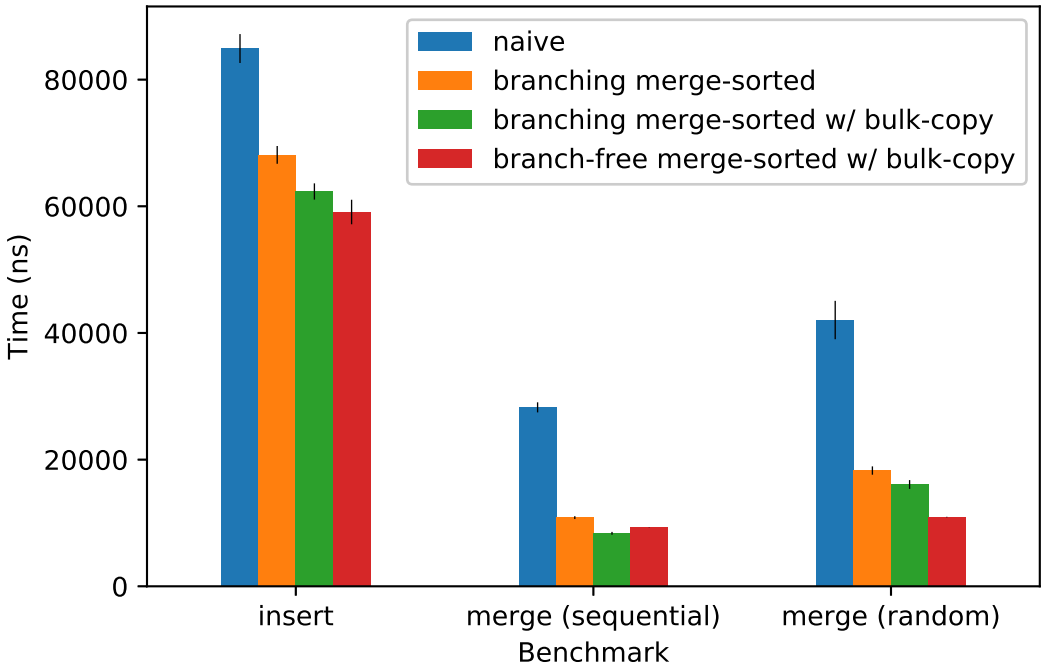

Figure 3.2: Benchmarks for KLL implementations

This observation leads to a third optimization: merge the sorted sequences using the branch-free algorithm in Listing 3.4. At each iteration, two masks are calculated, one for the first value and one for the second value. Exactly one of those

masks will be set to all ones, and the other will be set to all zeroes. The selected value can then be calculated using bitwise AND and OR operations. This is functionally equivalent to the simple merge-sorted algorithm, except that the CPU always executes the same instructions on each iteration of the loop.

Listing 3.4: Function that merges two sorted sequences without conditional branches

```
fn merge_sorted(v1: &[u32], v2: &[u32]) -> Vec<u32> {
    let (n, m) = (v1.len(), v2.len());
    let mut result = Vec::with_capacity(n + m);
    let (mut i, mut j) = (0, 0);
    while i < n && j < m {
        let lt = v1[i] < v2[j];
        let v1_mask = !(lt as u32).wrapping_sub(1);
        let v2_mask = !(!lt as u32).wrapping_sub(1);
        let val = (v1[i] & v1_mask) | (v2[j] & v2_mask);
        result.push(val);
        i += lt as usize;
        j += !lt as usize;
    }
    result.extend_from_slice(&v1[i..n]);
    result.extend_from_slice(&v2[j..m]);
    result
}
```

In Figure 3.2 we see that the branch-free implementation results in a 31.9% decrease in runtime compared to the branching implementation when merging random data. However, the branch-free implementation is 11% slower on the sequential data benchmark, since the CPU can accurately predict the branches taken in that benchmark. For the application monitoring case, this trade-off makes sense, since the system cannot know *a priori* whether the input data will cause the merge operation to branch predictably.[7]

Taken together, these optimizations significantly improve the performance of the KLL sketch for insert and merge operations. The next section empirically evalu-

[7]It may be possible to develop heuristics for when the branching implementation might outperform the branch-free implementation. If so, the merge algorithm could choose the implementation based on these heuristics. This would be an interesting direction for future work.

ates the performance of the branch-free implementation in greater detail.

3.4. Performance Evaluation

While the KLL sketch provide strong theoretical guarantees, its performance in practice is less well understood. This section documents the results of several experiments designed to evaluate the approximation error, space usage, and execution time of KLL sketches.

The parameter k is the capacity of the top-level compactor, with the other compactor capacities calculated as an exponential decay of k. We are interested in how KLL sketches behave for different values of k.[8] In particular, we want to understand how k affects error in quantile approximations, the number of values retained by the sketch, and the execution time of insert, merge, and query operations.

3.4.1 Approximate Quantile Error and Sketch Size

Figure 3.3 shows the normalized rank error for sketches summarizing a dataset of $10,000$ values. The values were generated uniformly at random from 0 to $2^{32} - 1$, inclusive. The test executed 100 trials, each of which performed queries for each $\Phi = 0.1$, $\Phi = 0.2$, .., and $\Phi = 0.9$. The test then compared the sketch's estimate of the quantile to the exact quantile to calculate the normalized rank error.

We can see from Figure 3.3 that as k increases, the normalized rank error decreases. Moreover, the errors become more tightly distributed at higher values of k. This is expected—retaining more values from the input data stream allows the sketch to answer quantile queries with greater accuracy.

The sketch sizes are shown in Figure 3.4. We can see that sketch sizes increase

[8] Each of the tests set the capacity to k for the top $\left\lceil \log_2 \log_2 \frac{1}{\delta} \right\rceil = 5$ compactors. This corresponds to a failure probability of $\delta \approx 10^{-8}$.

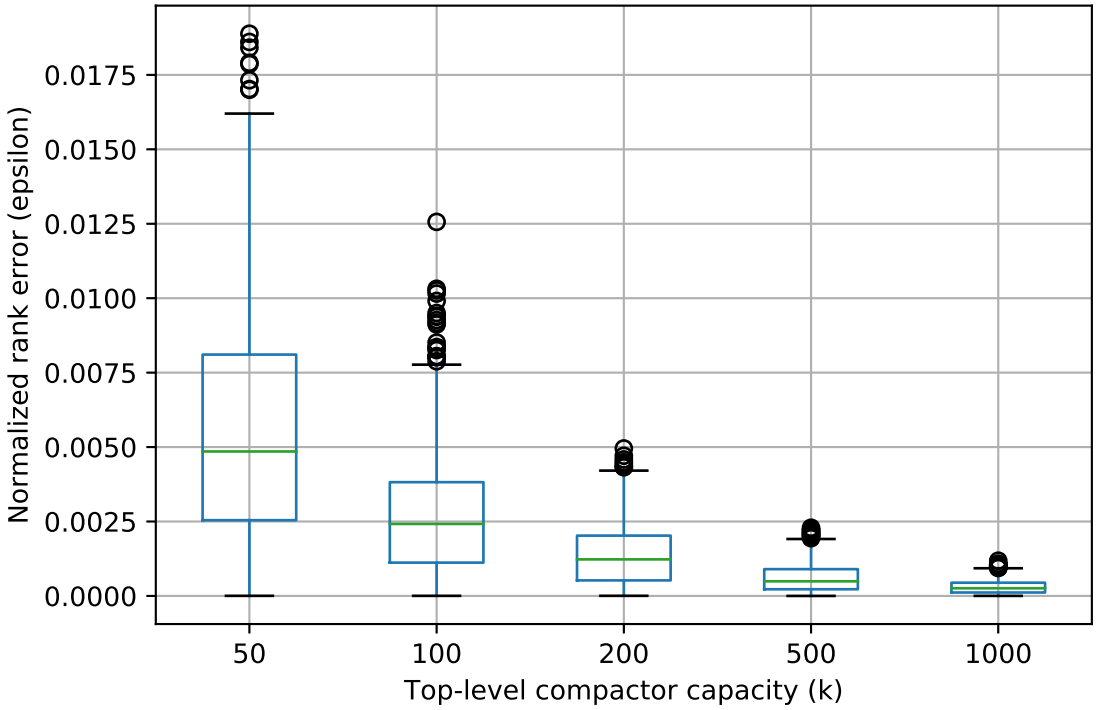

Figure 3.3: KLL sketch error boxplot

as more values are inserted. At intervals, the sketch sizes decrease due to compaction. Sketches with lower values of k perform more frequent compactions that each discard a small number of values; sketches with higher values of k perform fewer compactions that discard a larger number of values. Additionally, sketches with higher values of k tend to have larger sketch sizes because they can retain more values from the input stream.

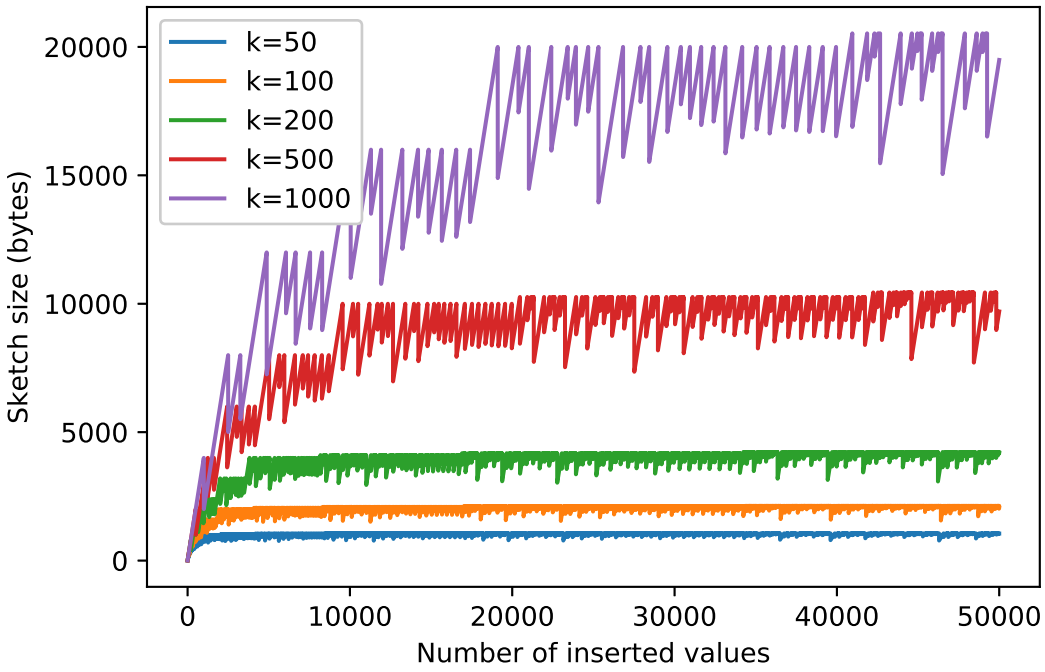

Figure 3.4: KLL sketch sizes

Recall that, in a KLL sketch, any compactor with minimum capacity is simulated by a random sampler in $O(1)$ space. One might wonder how much space this saves in practice. Table 3.3 compares the size of sketches with and without sampling enabled for an input data stream of one million values. We see a small saving in space for sketches with $k = 50$ and $k = 100$, but no savings at all for larger sketches. Why? In order for the sampler to take effect (sample rate $r \geq 2$), enough compactors must

have been added to the hierarchy that the lowest-level compactors have their capacity reduced to 2. Intuitively, sketches with smaller values of k "fill up" more quickly than sketches with larger values of k, so fewer inserts are necessary to trigger sampling. This suggests that while the random sampler improves the theoretical worst-case size of the sketch, the increase in code complexity may outweigh any practical benefits.[9]

k	**Size without sampler**	**Size with sampler**	**Difference**
50	1108	1040	68
100	2164	2136	28
200	4144	4144	0
500	10668	10668	0
1000	20832	20832	0

Table 3.3: Sketch sizes (in bytes) for different values of k built from a data stream of 1 million values

Overall, these data show that KLL sketches save significant space compared to the original dataset. Storing the full dataset of 1 million 32-bit values would require 4MB. In contrast, KLL sketches with $k \leq 1000$ reduce the required space by over 99%, while retaining the ability to answer quantile queries with high accuracy. Note that these results measure the size of the sketch without compression: each retained value uses 32 bits of memory.[10] In Chapter 4, we will see how to further reduce the size of the sketch using a lossless compression algorithm.

3.4.2 Merge and Query Execution Time

First, we consider the execution time of the merge operation. The test constructed two sketches—each from an input stream of length $n \leq 50000$—then mea-

[9] I suspect this is why neither Edo Lang's nor Yahoo!'s open source implementation of the KLL sketch includes the sampler (Lang, 2016) (Yahoo! Inc., 2018).

[10] Strictly speaking, the in-memory representation of the sketch may be larger, since the Rust vector implementation can allocate more memory than the number of inserted elements. However, when the sketch is serialized for transmission and storage, only the stored values are represented. In contrast, the open-source Yahoo! implementation uses a single array of fixed size, manually allocating additional memory as necessary (Yahoo! Inc., 2018). This results in a smaller in-memory sketch size, at the cost of additional complexity in the code.

sured the time to merge the two sketches, taking the median measurement across 10 trials. Figure 3.5 shows the results. We can see that greater values of k result in greater merge times, since more values must be merged. We can also see "spikes" in the execution times; these likely represent merges that triggered one or more compactions.

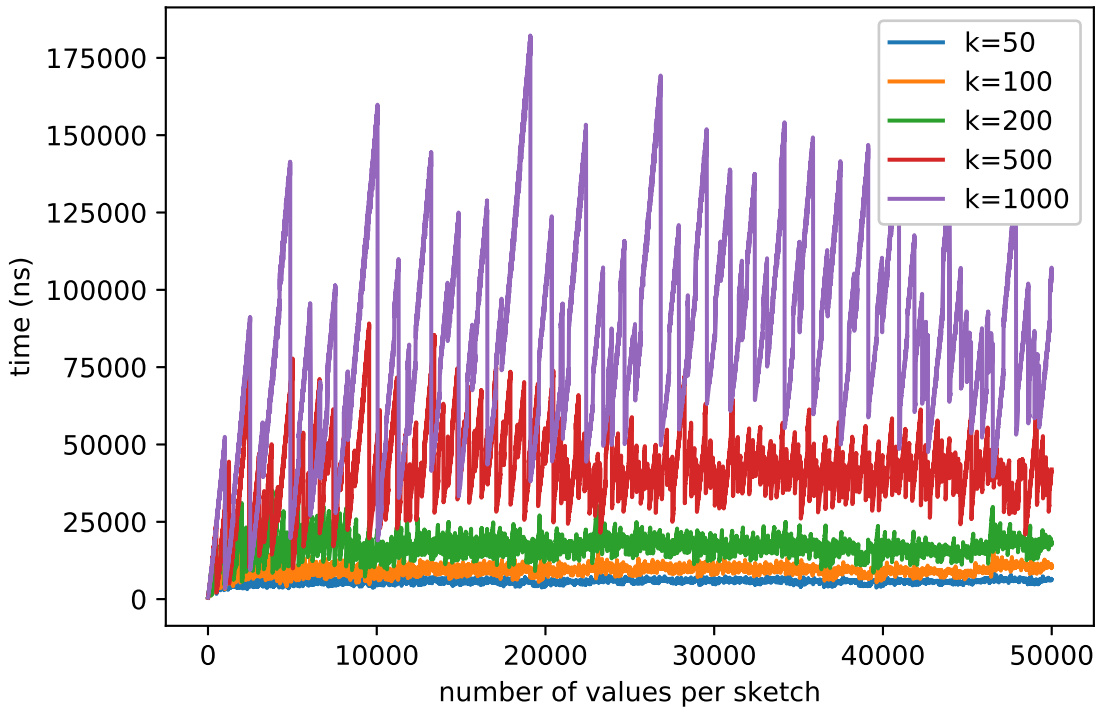

Figure 3.5: KLL sketch merge execution times

Similarly, Figure 3.6 shows the results from a test of query execution time. The test built a sketch from an input stream of n values, then measured the time to execute each of the quantile queries for $\Phi = 0.1$, $\Phi = 0.2$, ..., and $\Phi = 0.9$. No differences were observed between the measurements for different values of Φ, so for each n, the test treated each query as separate trial. In total, 90 trials were executed for each n (ten iterations, each executing nine queries), of which the median is reported.

As with the merge operation, we can see that sketches with lower values of k

take less time to execute quantile queries, since there are fewer stored values to sort and search. We also see "spikes" in the execution times similar to the graph of sketch sizes (Figure 3.5). This can be explained by compaction: there are fewer stored values immediately after compaction, so query execution time decreases.

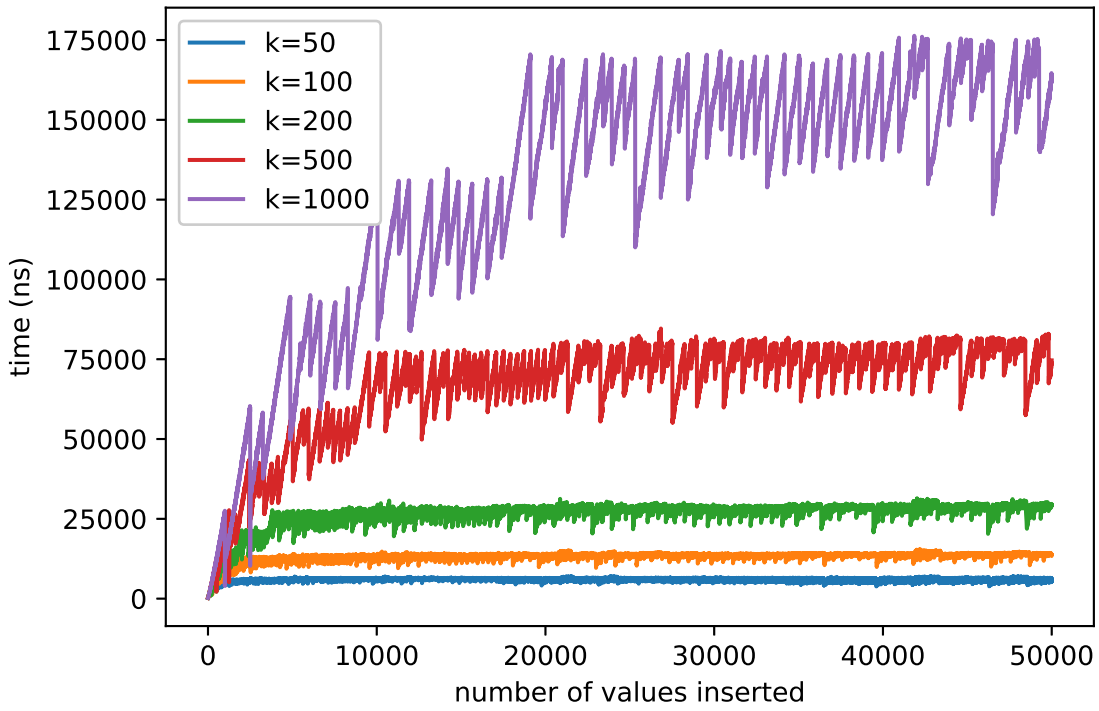

Figure 3.6: KLL sketch query execution times

Both merge and query execution time depend on the number of values stored in the sketch. How much of the execution time can be attributed to the sketch size? Table 3.4 shows the correlation between sketch size and execution time for the merge and query operations. The measured times for both operations are positively correlated with sketch size, but the query operation shows a stronger correlation. This makes sense because each query operation spends most of its time sorting and searching the stored values of the sketch. In contrast, the merge operation depends partly on *where* the values are stored in the compactor hierarchy. The merge operation

triggers a compaction only when too many values are stored at the same level of the hierarchy. Since the placement of values in the hierarchy is less closely tied to the overall size of the sketch, the merge operation's execution time is less strongly correlated with sketch size.

k	**Size and query time corr.**	**Size and merge time corr.**
50	0.7947274401463795	0.5763201116794751
100	0.9845953495773285	0.4536813844345805
200	0.9934287197214547	0.404897137426683
500	0.9976015020903097	0.4826670907757559
1000	0.9985575800165982	0.568108190062841

Table 3.4: Correlation between sketch size and execution times

Compared to the results in Gan et al. (2018), we see that the KLL sketch's merge execution time is slower than M-Sketch, which is consistently below 100 ns. However, the KLL query execution time is much faster than M-Sketch, which is greater than 10000 ns (and exceeds 1 ms for some datasets). This suggests that KLL sketches would perform better for user queries that involve relatively few merges, where the cost of the query operations dominates the cost of the merge operations.

3.4.3 Insert Execution Time and Throughput

Most insertions into a KLL sketch do not trigger compaction—the new value is simply written to the lowest-level compactor, an operation that takes only about 40 to 50 ns. However, the inserts that trigger compaction are more expensive, since the values from multiple compactors must be merged. Figure 3.7 shows quantiles of insert execution times for values of n from 1 to 1 million, taking the median measurement across ten trials. The graph shows the interaction between the *frequency* of compactions and the *cost* of each compaction. For smaller values of k, compactions are more frequent but less expensive. For example, sketches with $k = 50$ have more frequent compactions, so these times are included in the 75th percentile. In contrast,

more than 75% of inserts into sketches with $k = 1000$ do *not* trigger compactions. This explains why sketches with $k = 50$ have a higher 75th percentile time than sketches with $k = 1000$. The 99.9th percentile includes compactions for the $k = 1000$ sketch as well. Since these are more expensive than the compactions for the $k = 50$ sketch, the $k = 1000$ sketch dominates at the 99.9th percentile.

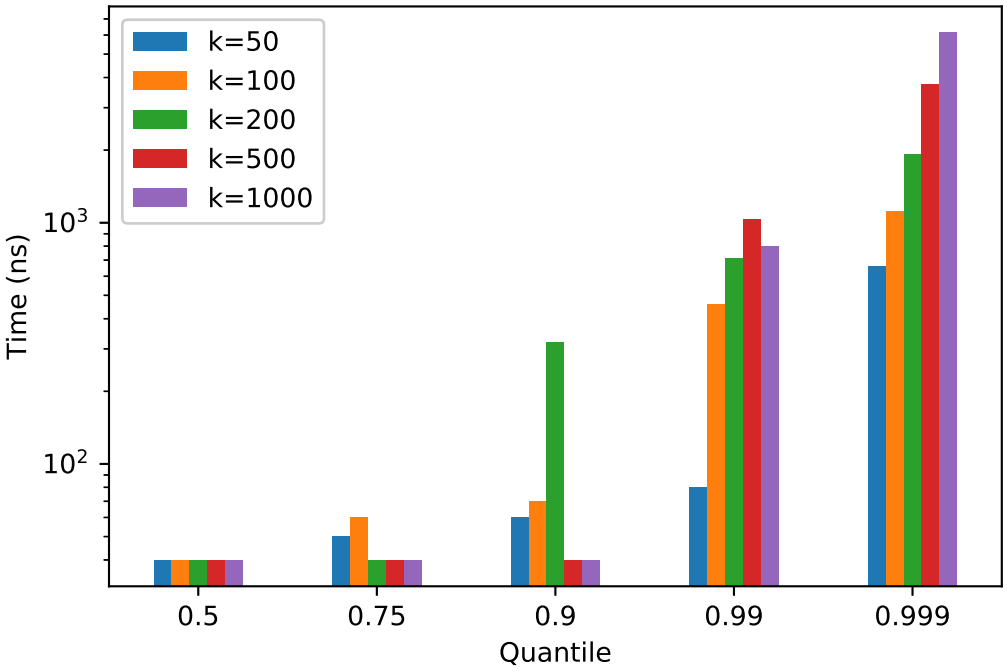

Figure 3.7: Quantiles of KLL sketch insert execution times

One might wonder how compaction frequency and cost affect the throughput of inserts into the sketch. This is important for the application monitoring case, since the sketch needs to "keep up" with the data points produced by the application. Is it better to have many cheap compactions or few expensive compactions? Figure 3.8 shows the throughput for sketches with different values of k, both with and without the sampler. First, consider the throughput for sketches that do *not* use a sampler. We can see that throughput increases for larger values of k. Even though sketches

with larger values of k perform more expensive compactions, they do so less frequently, so these sketches can process more inserts per second. With the sampler disabled, sketches can achieve higher throughput by performing fewer and more expensive compactions.

Now consider the throughput when the sampler is enabled. One impact of sampling is that it reduces the number of compactions for sketches with smaller values of k. The lowest-level compactors from these sketches are absorbed into the sampler, so compaction at the lowest levels is simulated using reservoir sampling. Since reservoir sampling is faster than compaction, we see increased throughput from sketches with $k = 50$ and $k = 100$. However, the sampler has no effect for sketches with greater values of k because, as we observed when comparing sketch sizes, 1 million inserts are not enough to trigger sampling. Therefore, with the sampler enabled, sketches with smaller values of k can reduce the number of compactions, thus resulting in higher throughput.

Overall, these results show that KLL sketches are competitive with the sketches surveyed by Wang et al. (2013). That study found that MRL99 was the mergeable sketch with the fastest insert rate, achieving an average insert time of less than 200 ns for a sketch with 10^{-6} average error. Table 3.5 shows comparable results for the KLL sketch, suggesting that KLL sketches would be a good candidate for the application monitoring system proposed in Chapter 2.

k	**Average error**	**Average insert time (ns)**
50	0.005592	48.209475
100	0.002746	66.452718
200	0.001380	94.865806
500	0.000601	74.692325
1000	0.000309	71.684154

Table 3.5: Average error and insert times for KLL sketches

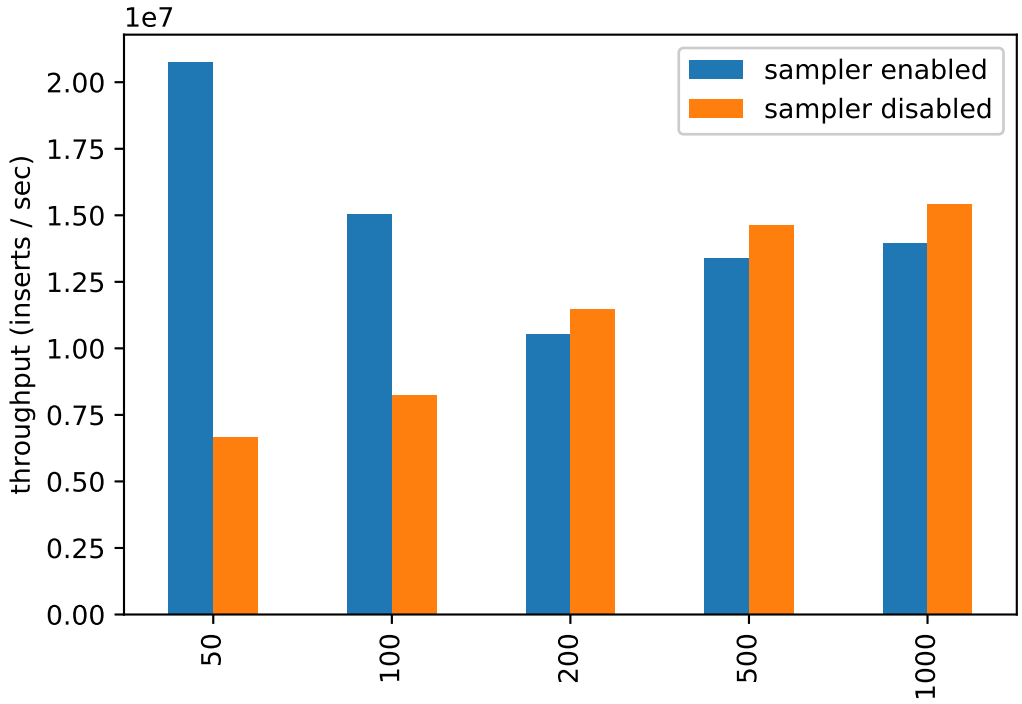

Figure 3.8: KLL insert throughput

3.5. Conclusion

In this chapter, I argued that KLL sketches show potential for use in the application monitoring system presented in Chapter 2. The experimental data show that an optimized KLL sketch greatly reduces the required space compared to the original dataset while retaining the ability to answer quantile queries with small error. Additionally, we saw that the insert, merge, and query operations perform well, at least on small-scale benchmarks.

In Chapter 6 we will evaluate how well KLL sketches work in the context of a full application monitoring system. To understand that system, however, one must first understand how the system can use individual quantile sketches to answer user queries (Chapter 5) and how quantile sketches can be compressed and stored on disk (Chapter 4). We turn to those topics now.

Chapter 4: Storage and Compression

This chapter describes how quantile sketches are stored by the application monitoring system proposed in Chapter 2. First, section 4.1 describes how sketches can be persisted in a key-value storage engine, focusing on the RocksDB embedded database developed by Facebook. Next, we consider how to optimize the disk usage of the application monitoring system. As we saw in Chapter 3, the KLL sketch greatly reduces the number of data points the system needs to store. However, two techniques can improve disk usage even further. Section 4.2 describes how older sketches can be merged in a process called "downsampling," and section 4.3 shows how to further compress the data stored in each sketch.

4.1. Storage Engine

A *key-value store* is a software component that stores and retrieves values using a lookup key. Keys are strings, and values are arbitrary byte arrays. Keys are always totally ordered, but the ordering can be configured based on the needs of the application. For our purposes, the key-value store must support four core operations:

- `set(`k, v`)` sets a value v for a key k.
- `get(`k`)` retrieves a value for the key k
- `getrange(`k_1, k_2`)` retrieves all key value pairs (k, v) where $k_1 \leq k \leq k_2$. Either key can be set to null, which is interpreted as an infinite bound on the requested range.

- `delete(`k`)` deletes the value associated with the key k.

Furthermore, to support the downsampling process described in section 4.2, the store must support transactions, thus allowing a series of `set` and `delete` operations to execute atomically and in isolation.[1] Finally, the store must persist data to disk.

Examples of key-value stores that meet these criteria include BerkeleyDB (Seltzer & Bostic, 2011), Apache Cassandra (Lakshman & Malik, 2010), Amazon's DynamoDB (DeCandia et al., 2007), and RocksDB (Facebook Engineering Team, 2017). Some stores, such as Cassandra and DynamoDB, are deployed as independent network services; others, such as BerkeleyDB and RocksDB, are embedded as libraries. The remainder of this section focuses on RocksDB, because it is easy to embed in the proof-of-concept system evaluated in Chapter 6. Additionally, as we will see shortly, RocksDB provides a "merge operator" feature that simplifies the interactions between the backend instance and the storage engine. However, in a production-ready system, Cassandra or DynamoDB would likely prove more reliable; unlike RocksDB, these can scale horizontally to provide additional storage, read/write throughput, and high-availability.

RocksDB is an embedded key-value store based on the LSM tree data-structure (Facebook Engineering Team, 2017). LSM trees are designed to batch writes to disk[2] while supporting both point- and range-queries (O'Neil et al., 1996). Any change to a key-value pair (insertion, update, or deletion) is initially written to an in-memory buffer; when the buffer reaches capacity, the changes are sorted by key and flushed to secondary storage. Sorted runs of changes in secondary storage are organized into levels of increasing capacity. When a level fills, its sorted runs are merged into the

[1] Specifically, the transaction must provide at least the read-committed isolation level to prevent dirty reads.

[2] Another tree data structure optimized for write throughput is called a fractal tree (also known as a streaming B-tree) (Kuszmaul, 2010). Like an LSM tree, fractal trees batch changes into a buffer, merging sorted sequences to disk in a background thread to achieve high write throughput.

runs at the next, larger level. The process that migrates changes through the level hierarchy is called "compaction" (not to be confused with the KLL sketch compaction we saw in Chapter 3). To query an LSM tree, one searches for the most recent version of a given key-value pair, starting with the in-memory buffer, then continuing through each level from smallest (newest data) to largest (oldest data) (Dayan & Idreos, 2018). LSM trees are well-suited for the application monitoring case, because (a) they support high-write throughput and (b) they perform better when querying recent data than when querying older data. However, the backend instance in our system does not require that the key-value store use LSM trees, as long as the store supports the operations described above.

When the daemon sends a sketch to the backend instance, the backend instance must persist the sketch to disk. Recall from Chapter 2 that each sketch represents a window in a series. The window is a pair of timestamps indicating the start and end of the window's interval, and the series is a string that the application uses to identify the measurements. How does the backend instance interact with the key-value store to persist the sketch?

First, the backend instance constructs a key for the sketch. The key is a concatenation of the series string and the window's start timestamp. The value is the window's end timestamp, followed by the sketch data. Keys are totally ordered, first by the series string and then by the start timestamp. The system relies on this ordering of keys to efficiently retrieve sketches from the store.[3]

When the backend instance inserts a sketch into the key-value store, a sketch may already exist with the same key. This can happen, for instance, if two separate

[3]Notice that the key structure ensures that at most one window exists in a series starting at a particular time. However, it is possible for windows to overlap if the end timestamp from one window in a series is greater than the start timestamp of another window in the same series. As we will see in Chapter 5, the query subsystem allows overlapping windows to be returned in response to user queries.

application instances collect data for the same series at the same time. In that case, the new sketch must be merged with the stored sketch. RocksDB provides a convenient mechanism to achieve this called "merge operators." The backend instance can configure RocksDB to call custom code to merge a new value with an old value, if one already exists for the same key. For our purposes, the custom merge operator simply updates the end timestamp to the greater of the two end timestamps, then merges the KLL sketches. The same functionality can be achieved with other key-value stores by trying to retrieve the key before setting the new value; however, this complicates the interactions with the key-value store.[4]

As we will see in Chapter 5, all user queries need to retrieve stored sketches. In particular, queries need to retrieve sketches filtered by series and *start* timestamp.[5] This translates to a sequence of `getrange`(k_1, k_2) key-value store operations. Suppose a user queries the system for quantiles in the series s in the time range from t_1 to t_2. Since the keys in the key-value store include both the series name and start timestamp, and since these keys are totally ordered, the backend instance can call `getrange(key(`s, t_1`), key(`s, t_2`))`, where `key(`$\cdot$`)` constructs a key string from the series string and start timestamp. This will efficiently return all and only sketches with the correct series and start timestamp.

This architecture cleanly separates the core key-value operations from the rest of the application monitoring system. The key-value store itself communicates only in terms of string keys and byte-array values; it can be agnostic to the content of those keys and values. As a result, the system can leverage existing key-value stores

[4]Additionally, for key-value stores exposed over a network, the `get(`k`)` call adds the overhead of an additional network round-trip.

[5]It is possible that these queries will exclude a sketch that overlaps with the start of the requested range, but whose start timestamp is before the start of the requested range. This is acceptable in the application monitoring case because most sketches will have short duration (10 seconds by default) and represent disjoint time intervals. In this context, excluding one sketch at the start of the requested range will not prevent users from understanding the behavior of the system.

such as RocksDB to handle persistence and retrieval of sketch data.

4.2. Downsampling

In Chapter 2, we saw that many application monitoring systems support *downsampling*: the ability to combine older time windows in order to reduce disk usage. This section explains how downsampling can be implemented by merging quantile sketches, relying on the guarantees of the merge operation to reduce size while bounding approximation error.

Downsampling affects how the system responds to user queries. Suppose a user queries for the median of each window for a particular time series.[6] For recent data, the system would report medians for 5-second windows. Data older than five minutes would be reported in 10-second windows, and data older than a day would be reported in 1-minute windows. Since recent data is more valuable than older data in application monitoring, this is an acceptable trade-off to reduce storage costs.

To explain the downsampling process, we need a few definitions. First, the start and end of each window are represented as *timestamps*, which are 64-bit unsigned integers representing the number of seconds since the Unix epoch. Let $start(W)$ and $end(W)$ denote the start and end, respectively, of a window W. The *duration* of a window W is $end(W) - start(W)$. Every valid window has $start(W) < end(W)$; therefore, the minimum window duration is one second. We call the window *aligned* at a resolution $r > 0$ when $start(W) = kr$ for some integer $k \geq 0$ and $duration(W) \geq r$. For example, if $start(W) = 15$ and $end(W) = 21$, then W is aligned for $r = 5$, but not for $r = 10$. In contrast, if $start(W') = 30$ and $end(W') = 41$, then W' is aligned for both $r = 5$ and $r = 10$.

[6] In Chapter 5, we will consider other queries that the system can support.

The downsampling process runs in a background thread that scans each window stored in the database. Windows are downsampled based on a user-configured data retention policy such as the one in Table 4.1. For each window, the policy determines which of three downsampling actions to perform:

- **Ignore**: Do not modify the window.
- **Expand to resolution** r: Let t_1 and t_2 denote the start and end timestamps of the window, respectively. Then, update the start timestamp to $t_1' = \lfloor t_1/r \rfloor \cdot r$ and the end timestamp to $t_2' = \max(t_1' + r, t_2)$. The resulting window will be aligned at resolution r.
- **Discard**: Delete the window.

If start time is greater than...	**Then**
5 minutes	Expand to resolution 10
24 hours	Expand to resolution 60 (1 minute)
7 days	Expand to resolution 600 (10 minutes)
28 days	Expand to resolution 3600 (1 hour)
365 days	Discard

Table 4.1: Downsampling data retention policy

How can the downsampling process be implemented using the key-value store operations from section 4.1? First, the downsampling process must scan all keys in the database, which translates to a `getrange(`k_1, k_2`)` operation with both keys set to null (no start/end limits).[7] Second, when expanding a window, the downsampling process must delete one key (the old start timestamp) and insert another (the new start timestamp). This translates to a `delete(`k`)` and a `set(`k, v`)` operation, executed

[7]One might worry about the possibility of reading stale data during the scan. However, this presents a problem only if the stale data is used to overwrite newer data. Since newer windows are always ignored by the downsampling process, the system can avoid this problem by rejecting incoming windows that start early enough to be downsampled.

together as a transaction to avoid read anomalies. Note that if a given key already exists, as during sketch insertion, the custom merge operator will merge the new sketch with the existing sketch. Finally, discarding a window translates to a single `delete(`k`)` operation.[8]

The following example illustrates how the downsampling process works. Suppose two windows, W_1 and W_2, are inserted into the same series, where:

- $start(W_1) = 27$ and $end(W_1) = 32$
- $start(W_2) = 29$ and $end(W_2) = 34$

Based on the policy in Table 4.1, after five minutes have passed, the downsampling process would expand both W_1 and W_2 to align with $r = 10$. Both W_1 and W_2 would then start at 20 and end at 40. Since both windows have the same start timestamp and series string, the storage engine would merge the two windows into a single window. Similarly, the downsampling process would expand the windows again after 24 hours, 7 days, and 28 days. Finally, after a year had passed, the windows would be discarded.

Downsampling can significantly reduce the storage footprint of the system. Figure 4.1 shows the effects of downsampling over time. Initially, sketches were inserted for 10,000 time series, each with five minutes worth of data in 5-second windows. In total, the sketches represented 1 billion inserted datapoints, although only a fraction of these are retained in the stored sketches. We can see that by expanding and merging windows, the system can reduce its disk usage. After five minutes, the data size decreases by 67%; after one day, the size decreases further to 27%. Due to

[8]One challenge with downsampling in a storage engine based on LSM trees, such as RocksDB, is that each update and deletion creates a new change record, temporarily increasing the size of the database. The storage engine reclaims the space occupied by the superseded or deleted key-value pair during compaction. In practice, this can cause operational instability if a large number of windows are merged or discarded during downsampling. One can mitigate such issues by carefully configuring RocksDB so it compacts before exceeding the available disk space.

the LSM compaction algorithm, some deletion change records continue to use disk space, since their total size does not exceed the threshold to trigger compaction. This accounts for the remaining 15MB disk usage after all windows have been discarded.

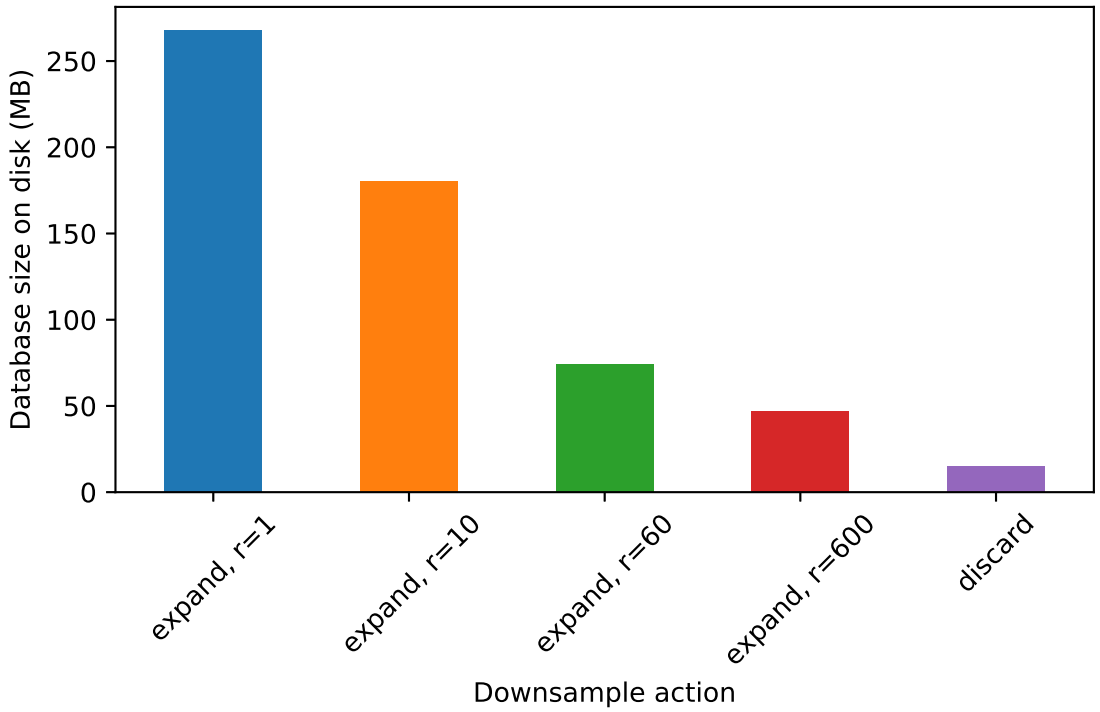

Figure 4.1: Effect of downsampling on disk usage

As we saw in Chapter 2, other application monitoring systems support downsampling as well. However, these systems use statistical summaries that are not mergeable. When such systems merge summaries during downsampling, they provide no guarantee about the accuracy of quantile queries. In effect, these systems give up both precision and accuracy, whereas a system based on mergeable quantile sketches loses precision but maintains ϵ-approximate accuracy guarantees.

4.3. Lossless Compression

In Chapter 3, we saw that each KLL sketch retains a number of values from its input data stream. The in-memory representation of the sketch stores the values for each compactor in a vector of unsigned 32-bit integers. The vector data structure allows the sketch to efficiently insert new values and perform compactions. However, we can further reduce the size of the sketch by compressing the values in each compactor. This is useful when the sketch is transmitted over the network or persisted to disk for two reasons. First, network and disk writes are usually I/O bound, so reducing the number of bytes written can potentially improve network and disk write throughput. Second, disk space costs money—either the up-front cost of buying hardware or an on-going cost of renting from a cloud provider—so compression might lower the system's operational cost.

At the same time, the system must be able to compress and decompress sketches quickly. In the architecture proposed in Chapter 2, sketches are sent periodically from the daemon to the backend instance, which then writes the sketches to disk. To reduce network bandwidth, the daemon could compress the sketch before sending it to the backend. However, it is important that the daemon "keep up" with the datapoints sent by the application, otherwise its memory usage could increase at an unsustainable rate and recent data would be unavailable for user queries. For this reason, sketch compression must not introduce too much overhead.

Decompression must also be efficient. When responding to a user query, the backend instance retrieves sketches from disk. Since the sketches are stored in a compressed format, the backend first decompresses the sketches so the sketches can be merged and queried.[9] As we will see in Chapter 5, a single user query might

[9]It may be possible to find a compression scheme that allows one to merge and query the com-

require the retrieval of many sketches. Decompression must execute quickly to avoid negatively impacting query response times.

We therefore need a compression scheme that can compress and decompress sequences of integers with minimal overhead. To preserve the error guarantees of the KLL sketch, the compression scheme must be lossless. While both space savings and runtime efficiency are important, the latter more directly impacts the system's performance. This leads us to consider compression schemes that can compress/decompress at high throughput, even if they result in slightly larger representations of the sketches.

Each compactor in the KLL sketch stores a sequence of values. For convenience, let us assume that the values are unsigned 32-bit integers. In Chapter 3, we introduce the invariant that each compactor (except for the first one) store its values in sorted order. Our goal then was to optimize the execution time of the compaction operation, but this invariant has benefits for compression as well. After sorting the values in the first compactor, every compactor in the KLL sketch stores a sequence of sorted integers. The need to compress a sequence of sorted integers arises in many data systems, including document search engines and social network graphs. As a result, many techniques have been developed to compress such sequences efficiently.[10] For our purposes, we focus on the Stream VByte approach developed by Lemire et al. (2018), since it can decode integers at high throughput (billions per second).

Stream VByte is a byte-oriented encoding scheme. The key idea of all byte-oriented encoding schemes is that small integers can be encoded in fewer bits than large integers. Unfortunately, the values stored in a KLL sketch might not be small

pressed sketch directly, in the spirit of the approach Abadi et al. (2006) describe for analytical queries on column-stores. Such an approach might improve performance by reducing the amount of data that needs to be transferred from RAM to the CPU caches. This could be an interesting direction for future work.

[10]For a recent survey, see Pibiri & Venturini (2018).

integers. However, since the integers are sorted and unsigned, the *differences* between consecutive values will always be less than or equal to the values themselves. A technique called *delta encoding* exploits this fact by subtracting consecutive values, then byte-encoding the resulting differences.

Stream VByte uses SIMD (single instruction, multiple data) instructions to decode multiple deltas in parallel. The encoded representation consists of two sections: a stream of "control" bytes, followed by a stream of byte-encoded integers. Each control byte maps to a *byte length* and a *shuffle mask*. The decoder first reads a control byte, then reads the corresponding number of bytes from the encoded integer stream. Next, using a SIMD instruction on the encoded block and the shuffle mask, the decoder moves bytes from the integer stream into a block of 4 unsigned 32-bit integers representing the deltas. Finally, the previous value is added to each delta to produce the original sequence of integers. Since the code is branch-free and processes multiple values in parallel, Stream VByte achieves high decoding throughput.

Encoding is also efficient. To encode a block of 4 integers, Stream VByte first subtracts previous values to calculate the deltas. It then determines the encoded length for each delta. Smaller integers are assigned a shorter length. The four encoded lengths together determine the control byte for that block of integers. Finally, any remaining values that do not fit in a block of 4 are written (uncompressed) to the output stream. The whole encoding process can be implemented using simple arithmetic and bitwise operations without conditional branches.

How well does Stream VByte perform when compressing KLL sketches? We are interested both in the reduction in sketch size as well as encoding/decoding speed. As Lemire et al. (2018) show, both the compression ratio and speed of Stream VByte depend partly on the compressibility of the input data. Therefore, we need to test using both highly compressible and less compressible datasets.

The benchmark tests use synthetic data generated from the function $f(i) = 2^d \cdot i$, where i is the index of the value (starting from zero) and d is a non-negative integer. The domain of f is integers in $[0, 2^{32-d})$, limiting the range of f to a subset of the unsigned 32-bit integers. Small values of d produce highly compressible datasets, since the deltas between consecutive values are small; increasing d produces a less compressible dataset. Note that the synthetic dataset does not necessarily reflect the response time data we would expect from a real application. In practice, response time data can vary widely depending on the application workload, so it would be difficult to predict how the monitoring system would behave for any particular application. Instead, the benchmark tests aim only to show the range of possible sketch sizes and encoding/decoding overheads, with the assumption that real-world behavior would likely fall somewhere within this range.

Figure 4.2 shows how Stream VByte compression affects sketch size. The graph shows the size of a KLL sketch (with $k = 200$) for three synthetic datasets, compared to a baseline with compression disabled. We can see that, while the size of each sketch is linear in the number of stored values, compression reduces the size of the sketch by up to 63%, depending on the input data. This represents an average savings of up to 21.3 bits per stored value (Table 4.2).

How much does compression cost in terms of execution time? Figure 4.3 shows the time to encode a sketch, and Figure 4.4 shows the time to decode a sketch. Encoding serializes the sketch to bytes, including running Stream VByte to compress the values stored in each compactor; decoding produces a sketch from the encoded bytes. For comparison, the graphs show the baseline execution times for a sketch with compression disabled. From the graphs, we can see that Stream VByte introduces more overhead during compression than during decompression. Highly compressible datasets are slightly faster to encode; however, compressibility has only a small effect

on decoding time.

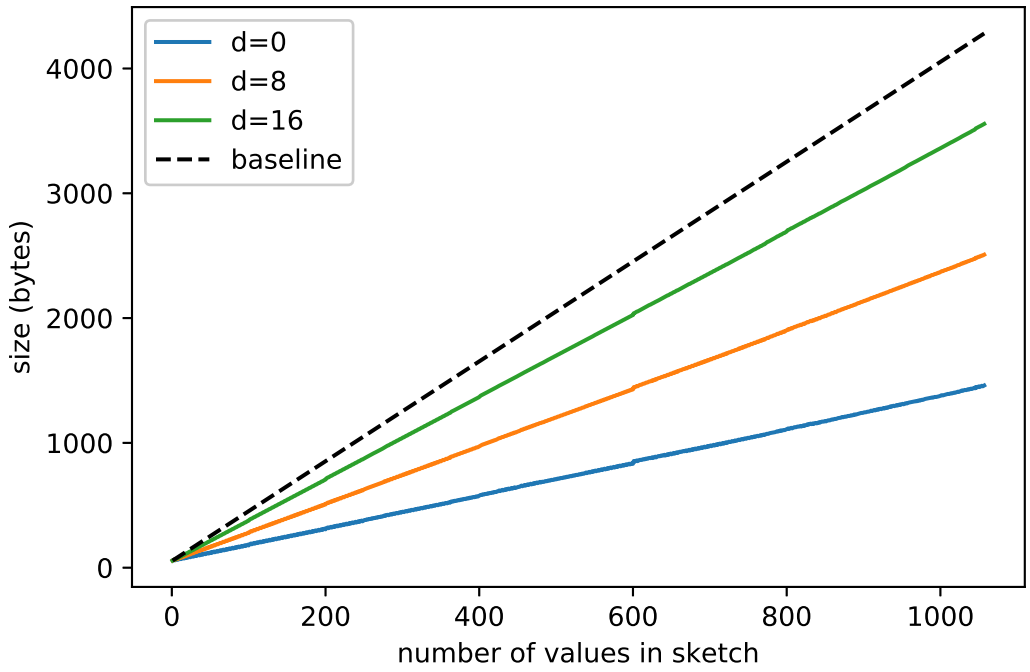

Figure 4.2: Compressed sketch size for synthetic datasets

	$d = 0$	$d = 8$	$d = 16$
avg sketch size decrease (%)	63.3	40.0	16.7
avg space saved (bits/value)	21.3	13.5	5.6

Table 4.2: Effect of Stream VByte compression on sketch size

These results show that the space saved by compression comes at the cost of greater execution times. However, compared to the sketch operations (see Chapter 3), the overhead of compression and decompression are minimal. Figure 4.5 shows the ratio between compression/decompression execution time and the insert/query/merge sketch operation.[11] The overhead of compressing a sketch is a fraction of the time to build the sketch, and the ratio drops to less than 1% after only a few hundred inserts.

[11]The ratios are calculated using the median execution times across all tested datasets. The ratio would be higher for less compressible data and lower for more compressible data.

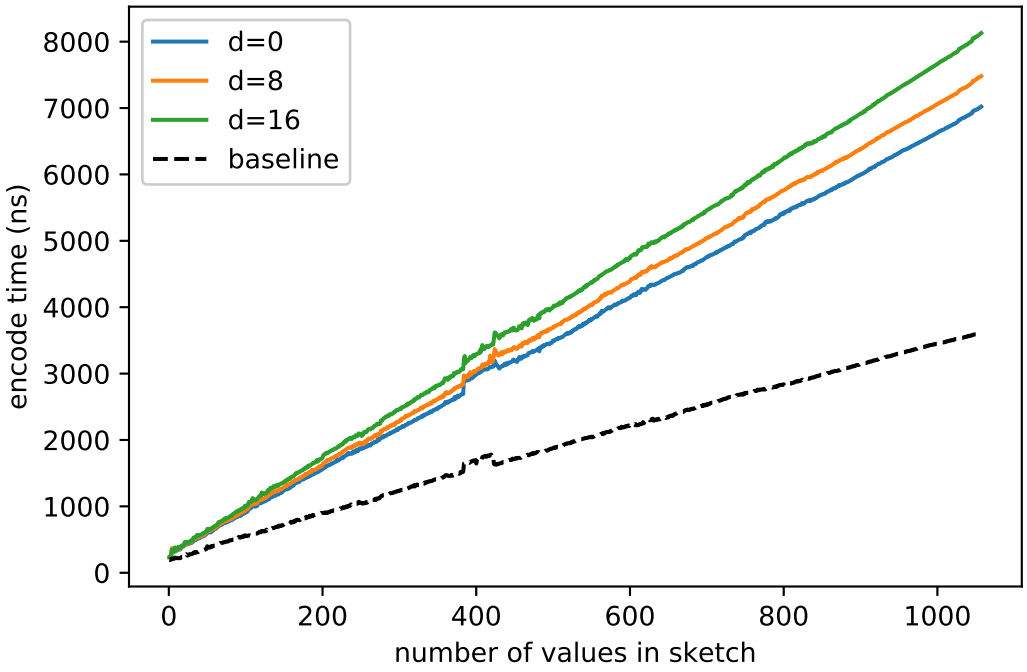

Figure 4.3: Time to encode a KLL sketch

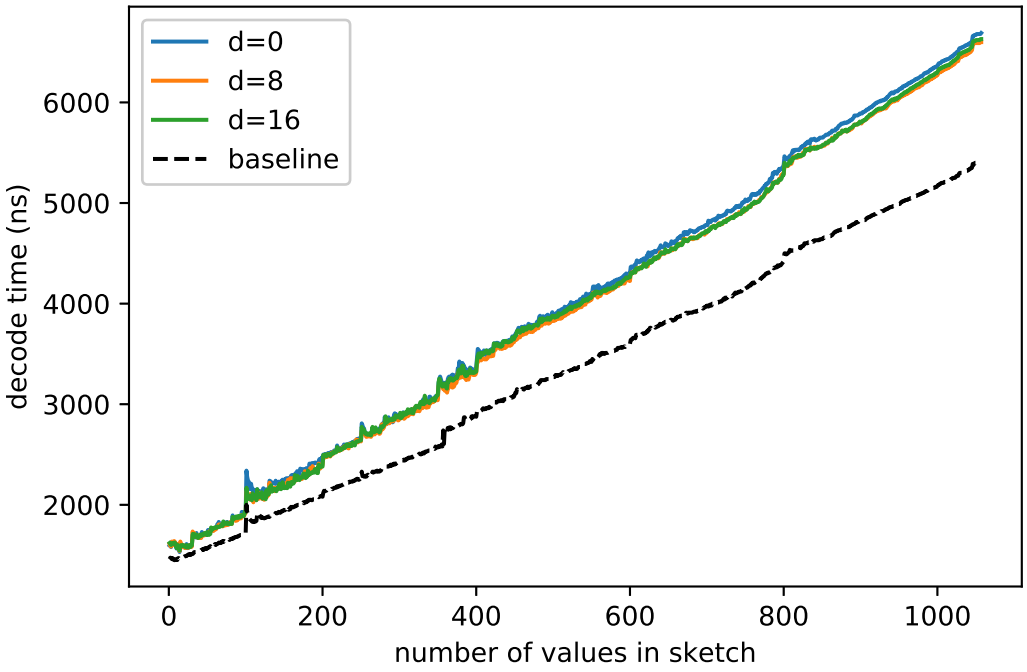

Figure 4.4: Time to decode a KLL sketch

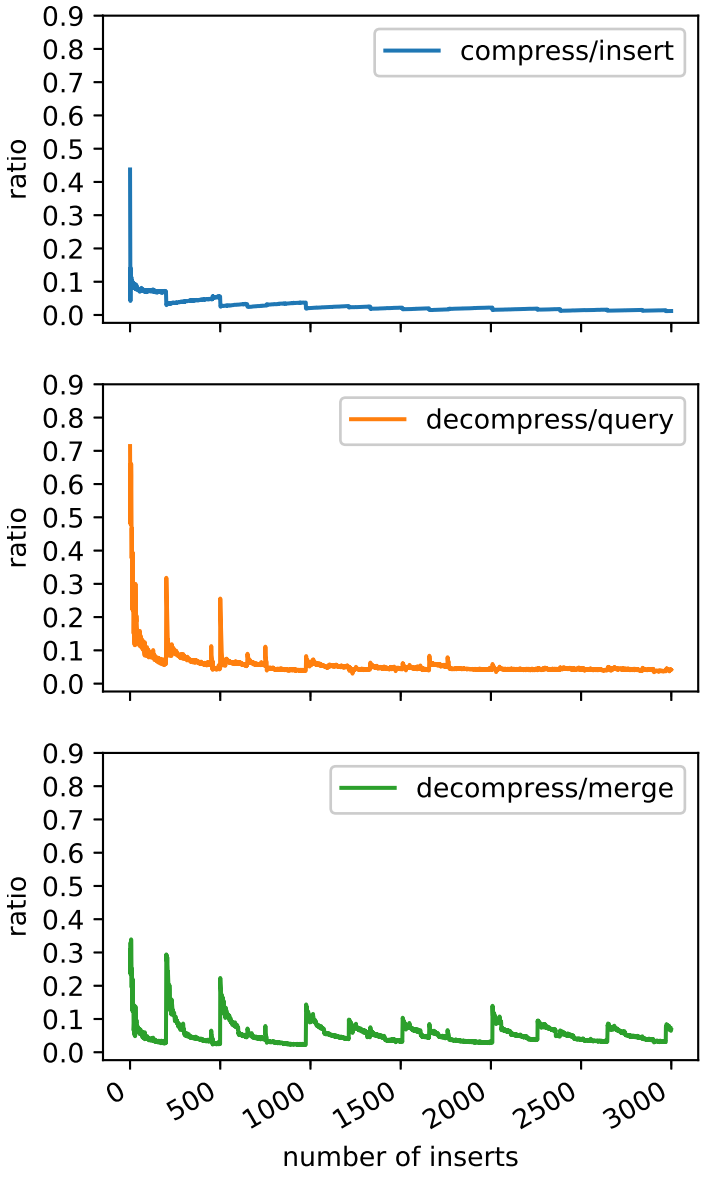

Figure 4.5: Execution time ratio of compression/decompression to KLL sketch operations

Likewise, decompression is much less expensive than the sketch merge and query operations. This suggests that the benefits of Stream VByte compression will usually be worth the additional overhead, at least for application monitoring use cases.

4.4. Conclusion

This chapter described how sketches are persisted and retrieved using a key-value storage engine. Additionally, we saw that the disk space needed to store these sketches can be significantly reduced through downsampling and lossless compression. Referring back to the architecture presented in Chapter 2, this fills in the details of how the backend instance stores the sketches it receives from the daemons. However, we have not yet seen how the system answers user queries. Retrieving sketches from the key-value store is only the first step in responding to user queries. The next chapter tells the rest of the story.

Chapter 5: Query Pipelines

In this chapter, we consider how the system proposed in Chapter 2 can answer queries from users. We have already seen that individual sketches can answer approximate quantile queries. However, this alone is rarely sufficient to allow human operators to diagnose production problems or for automated alerting systems to detect degraded application performance. To support these use cases, the system must aggregate sketch data to produce actionable information. Section 5.1 describes typical queries the application monitoring system should support. Section 5.2 then defines a set of query operators that can be composed into query pipelines. These pipelines merge sketches and calculate approximate quantiles to answer user queries.

5.1. Use Cases

What kinds of questions would a human operator ask when diagnosing a performance problem in a production system? As a starting point, let us consider the capabilities provided by Grafana, a popular open-source tool for visualizing operational data (Grafana Labs, 2018). Figure 5.1 is a screenshot from Grafana, representing 10 distinct time series from a fictitious e-commerce application. Clicking on one of the series hides the other series on the graph, as shown in Figure 5.2. Each point in the graph represents the 90th percentile response time for the "purchase" endpoint over a 10-second interval. This allows human operators to understand trends in the data over time. For example, during a spike in application usage, one would want

to understand whether the 90th percentile response time is increasing; if so, the application may be over-utilized. One would also want to understand trends across all 10 servers (each represented by a distinct time series), the servers in each region (US East and US West), and individual servers. For example, a spike in response times from the US West region could indicate a routing problem in a particular datacenter, whereas a spike in response time for only one server could indicate a misconfiguration on that server.

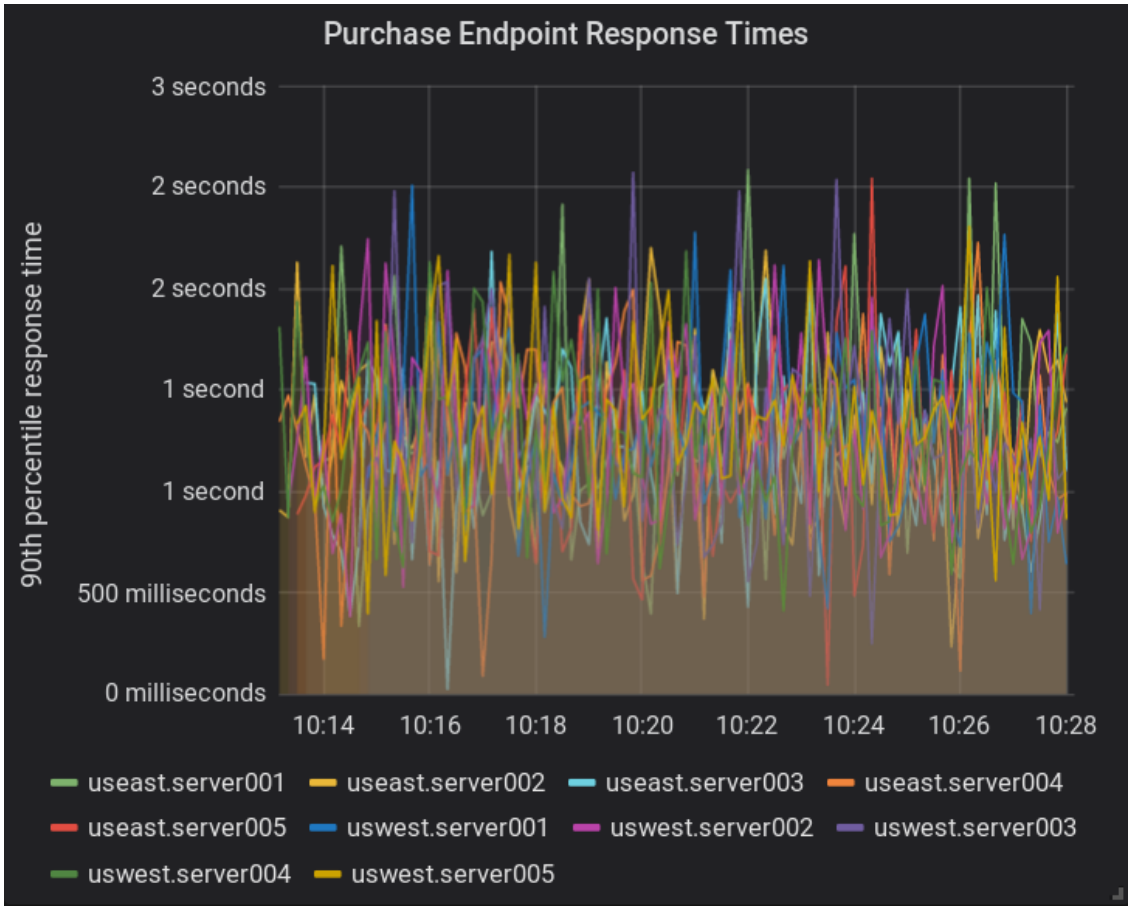

Figure 5.1: Grafana dashboard showing all time series

Many Grafana dashboards display important statistics, often linked to a service level agreement (SLA), as shown in Figure 5.3. For example, an SLA might impose financial penalties on a company if a particular endpoint's average response

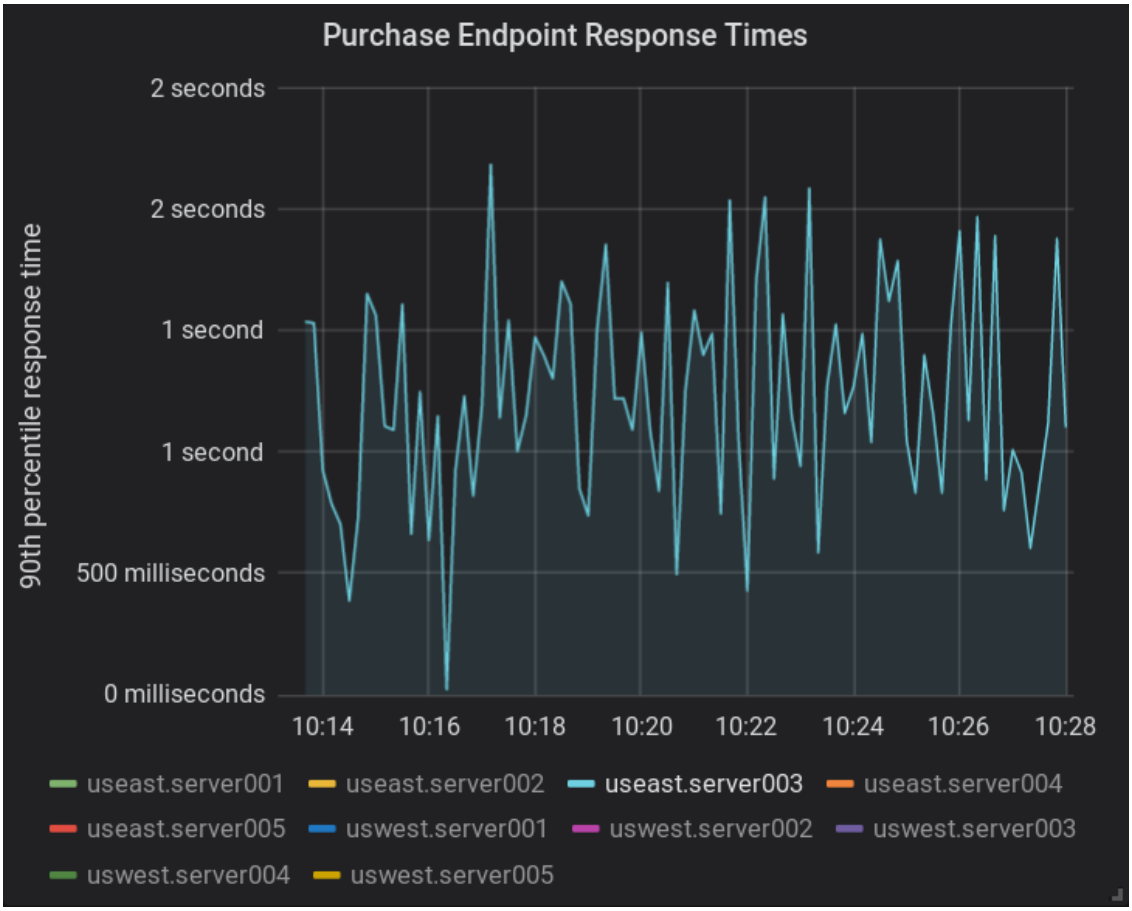

Figure 5.2: Grafana dashboard filtered to a single time series

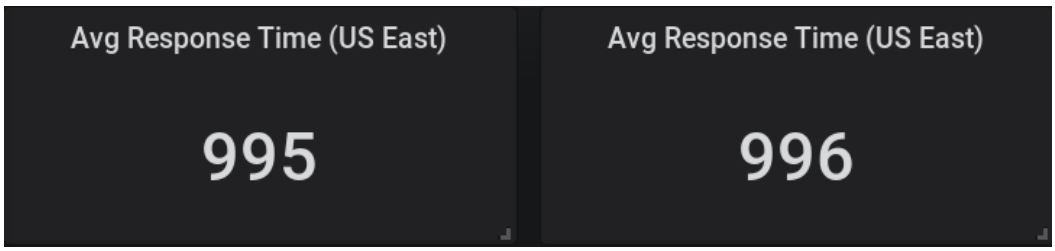

Figure 5.3: Grafana dashboard displaying summary statistics

time over the previous three hours exceeds a threshold in any geographic region. In this case, Grafana displays a single number for each region, aggregated over data from the previous three hours, answering the question, "Are we currently meeting the performance guarantees of the SLA?"

As we noted in Chapter 2, systems such as statsd limit the kinds of queries users can ask. We can see this limitation reflected in the Grafana dashboards when using a statsd-based data source. Figure 5.1 displays the 90th percentiles for each series individually, but it cannot display the aggregate 90th percentile across all series. Likewise, the time window for each measurement is always 10-seconds, since that is the interval at which statsd sends data into storage. For this reason, Grafana can display 10-second intervals, but it cannot aggregate these into larger intervals. Visually, it is difficult for human operators to compare, for example, how the 90th percentile across all servers in the last hour has changed compared to the same hour on the previous day.

From this discussion, we can identify several queries an application monitoring system should support, summarized in Table 5.1. Note that these queries are, at best, a subset of the kinds of queries a human operator might find useful.[1] However, even this subset includes and extends the functionality offered by Grafana, Graphite, and statsd, which have been widely used in the industry. Therefore, the remainder of this chapter focuses on how the system can answer these queries, setting the stage for the query response time tests in Chapter 6.

[1]For example, these queries do not include what Gan et al. (2018) call "threshold queries": find all series where the percentile exceeds some threshold.

Query	User Question
A	How has the 95th percentile response time changed for a single server over the last hour?
B	How has the 95th percentile response time changed across all servers over the last hour?
C	How does the 95th percentile response time over the last hour compare to each of the prior 24 hours?
D	What is the 95th percentile response time over the previous three hours?

Table 5.1: Application monitoring queries

5.2. Query Pipeline

To answer user queries, the system defines several *query operators* responsible for retrieving and transforming sketch data. Users can compose query operators into a *query pipeline*, which defines the flow of data from one operator to the next. At the start of the pipeline, an operator called `fetch` retrieves sketches from the storage engine we saw in Chapter 4. These sketches are (optionally) transformed by intermediate operators. Finally, at the end of the pipeline, the `quantile` operator approximates one or more quantiles for each incoming sketch. The resulting quantile approximations answer the user's query.

Table 5.2 shows the query pipelines for each of the queries we saw in the previous section. The system uses a functional notation to define query pipelines. For example, the pipeline that answers query A is defined as:

`quantile(fetch(`s`,` t_1`,` t_2`),` `0.95)`

This defines a pipeline with two operators: first, `fetch` retrieves sketches from storage for the series name s (a string) that start in the time range from t_1 to t_2 (integer timestamps). These sketches are then fed, one at a time, to the `quantile` operator.

The `quantile` operator, in turn, calculates the approximate 0.95-quantile for each incoming sketch, and outputs the approximate quantiles as the query result.

Query	Pipeline	Explanation
A	`quantile(fetch(`s`,` t_1`,` t_2`),` 0.95`)`	Approximate the 0.95-quantile for each time window in the series s that started between t_1 and t_2 (previous hour)
B	`quantile(combine(fetch(`s_1`), fetch(`s_2`)),` 0.95`)`	Merge overlapping windows from s_1 and s_2, then approximate the 0.95-quantile for each time window in the combined series
C	`quantile(group("hours", fetch(`s`)),` 0.95`)`	Merge windows from s that start in the same hour, then approximate the 0.95-quantile for each window
D	`quantile(coalesce(fetch(`s`)),` 0.95`)`	Merge all windows from s, then approximate 0.95-quantile for that window

Table 5.2: Examples of query pipelines

Listing 5.1 shows the Rust type definitions for a query operator and its outputs. Each query operator satisfies an iterator-like interface, allowing the caller to "pull" outputs from the operator. To run the pipeline for query A, one would pull the next

output from the `quantile` operator. The `quantile` operator would store a reference to a `fetch` operator, from which it would pull outputs (sketches). Note that `quantile` could have stored a reference to *any* operator that satisfies the common operator interface. Since all operators satisfy the same interface, one can compose operators into arbitrary pipelines to answer different user queries.[2]

All the queries in Table 5.2 can be answered using only five distinct query operators. Individually, each operator is simple, requiring between 29 and 201 lines of Rust code in my implementation. The next subsections describe each of these operators in detail.

5.2.1 Fetch

The `fetch` operator retrieves sketches from the storage engine. It accepts as arguments the series string, a start timestamp, and an end timestamp. The start timestamp and end timestamp are optional; if omitted, all sketches for the series will be returned. Most of the work occurs in the storage engine: the fetch operator simply maps the values returned by the storage engine's `getrange` operation to a sequence of sketches.

Since `fetch` is the only operator that does not depend on another operator, every pipeline must include at least one `fetch` operator as the "source" of data.

[2]There is an interesting parallel between how sketches are sent through the query pipeline and the "vectorized processing model" used by MonetDB for analytic workloads (Boncz et al., 2005). Like in MonetDB, each query operator outputs a vector of values at a time, namely the values stored by the KLL sketch. This may have performance advantages over "tuple-at-a-time" systems whose query operators output individual values (e.g. time series databases that store individual datapoints) by amortizing the cost of conditional jumps (through pointer indirection from one query operator to another) and better utilizing the CPU caches.

Listing 5.1: Type definitions for a query operator and its outputs

```
pub trait QueryOp {
    // Each query operator satisfies an interator-like interface
    // As an optimization, the outputs are returned *in order*
    // by starting timestamp
    fn get_next(&mut self) -> Result<OpOutput, QueryError>;
}

pub enum OpOutput {
    // Indicates that the operator has exhausted its outputs
    End,

    // Retrieved or transformed sketch
    // The sketch is called ‘‘writable’’ because it supports
    // the ‘‘merge’’ operator, which modifies the sketch’s data.
    Sketch(TimeWindow, WritableSketch),

    // Approximate quantile output by the pipeline
    // * The ‘f64‘ value is the requested ‘phi‘ value,
    //   to differentiate results if multiple values of ‘phi‘ were queried.
    // * The ‘Option<ApproxQuantile>‘ will be set to ‘None‘ only if
    //   the sketch has zero values, so no approximate quantile can be calculated.
    Quantile(TimeWindow, f64, Option<ApproxQuantile>)
}

// Unix timestamp = seconds since 1970-01-01T00:00:00Z
pub type TimeStamp = u64;

pub struct TimeWindow {
    start: TimeStamp,
    end: TimeStamp,
}

pub struct ApproxQuantile {
    // Number of inserted values
    pub count: usize,

    // Approximation of the quantile
    pub approx_value: u32
}
```

5.2.2 Quantile

The first argument to the `quantile` operator is an operator that produces a sequence of sketches, such as `fetch`. Each subsequent argument is a $\Phi \in (0, 1)$, indicating the quantile or quantiles to approximate. The operator outputs a sequence of (Φ, q) pairs, each representing an approximate Φ-quantile q for one of the sketches produced by the input operator.[3]

The `quantile` operator is the only operator that produces quantile approximations rather than sketches, so every pipeline must end with a `quantile` operator.

5.2.3 Coalesce

The `coalesce` operator merges all sketches produced by its input operator into a single sketch, then outputs that sketch (see Figure 5.4). For example, the `fetch` operator might retrieve all windows over the last 24 hours for a particular series. The `coalesce` operator could then merge these into a single aggregate sketch that it outputs to the `quantile` operator. The result would be an approximate quantile over the entire 24 hour period.

5.2.4 Combine

Whereas the `coalesce` operator merges sketches within the same series, the `combine` operator merges sketches from different series. The `combine` operator accepts one or more arguments, each of which is another operator that produces a sequence of sketches. It merges any sketches from its input operators that overlap, outputting a sequence of non-overlapping sketches, as shown in Figure 5.5.

For example, suppose that s_1 represents the response times for one server, and

[3] In functional programming terminology, the `quantile` operator might be called a "flat map" over the input sketches, calling the KLL sketch `query` operator for each requested value of Φ.

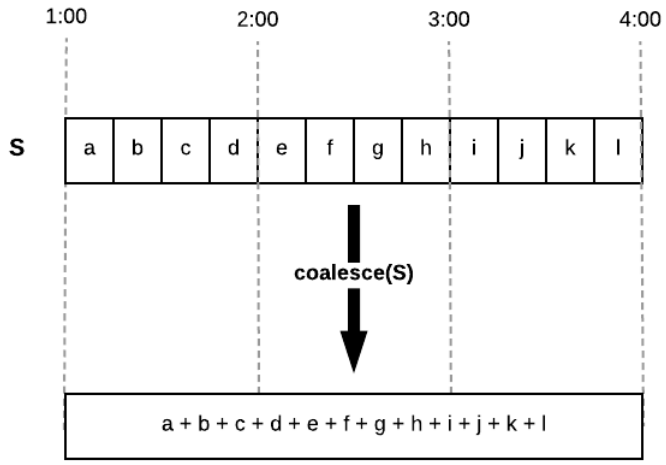

Figure 5.4: The `coalesce` operator merges all input time windows into a single merged window.

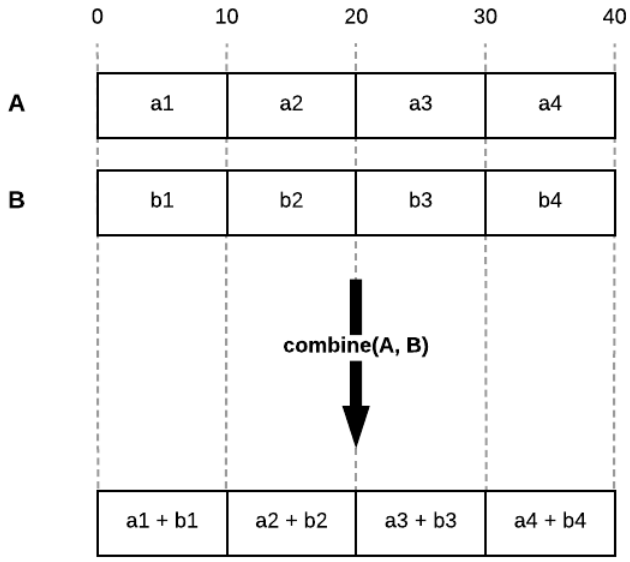

Figure 5.5: The `combine` operator merges overlapping time windows from two series, labelled **A** and **B**.

s_2 represents the response times for another server. Suppose further that both s_1 and s_2 contain sketches over the intervals $(0, 10)$ and $(10, 20)$. The `combine` operator would merge each sketch from s_1 with the overlapping sketch in s_2, outputting only two (non-overlapping) sketches. The result shows how the response time changes across *both* servers.

The `combine` operator can be implemented using a priority queue to minimize the number of sketches the operator must hold in memory at a given time. The sketch start time is used as the priority key, with earlier sketches output first. Since all input operators output sketches in ascending order by start time, the `combine` operator can enqueue the first sketch from each of its input operators. Whenever a sketch is popped from the priority queue, it is replaced by the next sketch (if any) from the same operator that originally produced the popped sketch. Overlapping sketches are therefore guaranteed to be adjacent in the priority queue.

The operator pops the first item from the priority queue and stores it in memory as the "current" sketch. If the next sketch in the queue overlaps the current sketch, the operator merges it into the current sketch; otherwise, it outputs the current sketch and replaces it with the next sketch in the queue. This continues until all input operators have been exhausted and the priority queue is empty. Using this approach, a `combine` operator with n input operators need hold only n sketches in memory at once.

5.2.5 Group

The `group` operator merges sketches that start in the same "time bucket," as depicted in Figure 5.6. Windows can be bucketed by hour or day. The result is a sequence of non-overlapping windows, where each window starts at either an hour boundary (if bucketing by hour) or a day boundary (if bucketing by day). For

simplicity, each day is defined to start at 12 a.m. UTC, but the system could be extended to support any time zone.

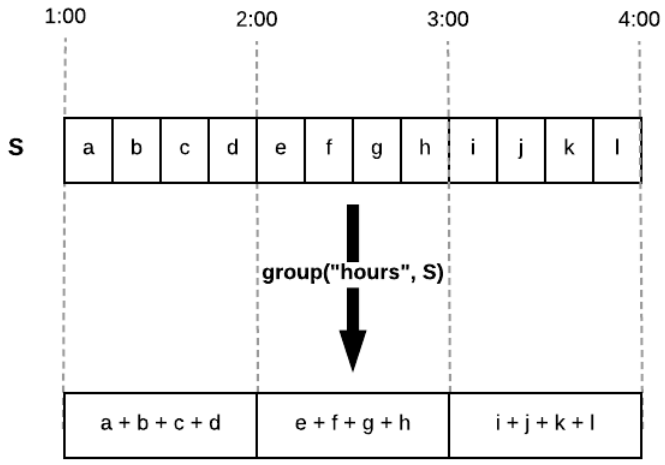

Figure 5.6: The `group` operator can be used to merge windows that start in the same hour.

The `group` operator makes it easier for human operators to interpret a dataset visually. For example, suppose a series has 360 sketches per hour, each representing a 10-second interval. When viewing a week's worth of data, one would see 60480 distinct sketches. With this many quantiles displayed, it can be difficult to see trends over the week. The `group` operator merges the sketches into 168 distinct buckets, one for each hour, allowing humans to more easily visualize trends over the entire week.

5.3. Conclusion

In this chapter, we saw that a small number of query operators organized into a query pipeline can provide useful information to human operators and automated alerting systems. With the query subsystem defined, we have filled in every part of the architecture proposed in Chapter 2. The question remains: how well does this

architecture perform at scale? The next chapter attempts to answer this question by testing a prototype system in a cloud environment.

Chapter 6: System Performance Tests

This chapter evaluates the performance of the design proposed in Chapter 2. Our goal is to understand how the system would perform under the real-world stresses of a production environment. In particular, this chapter evaluates the system along two dimensions:

- **Query Response Time**: How quickly does the system respond to user queries?
- **Write Throughput**: How many datapoints can the system ingest per second?

Section 6.1 describes the setup of the tests, including the details of the prototype. Then, Section 6.2 presents the results from three performance tests of the prototype running on Google Cloud Platform (GCP).

6.1. Test Setup

The prototype implementation, written in the Rust programming language, is open-source and available at https://github.com/wedaly/caesium. Following the architecture presented in Chapter 2, the prototype system consists of two kinds of network services: one or more *daemons* that receive datapoints from an application, and a *backend server* that stores sketches received from the daemons and answers user queries.

The daemon and server were configured to use one of two sketch implementations. One configuration used KLL sketches with $k = 200$. For comparison, a second "baseline" implementation was also tested. Unlike the KLL sketch implementation,

which discards datapoints to save space, the baseline implementation simply stores every datapoint. To isolate the impact of the KLL sketching algorithm rather than that of lossless compression, both the baseline and KLL implementations used Stream VByte (see Section 4.3) to compress the stored datapoints. Since both implementations satisfied the same mergeable sketch interface (insert, merge, encode/decode, and query), the daemon and server code could use either implementation interchangeably.

Both the server and daemon used non-blocking I/O to read bytes from the network in a single thread. On the server, the network messages were each enqueued in one of two bounded buffers, one for inserts and one for queries. Each buffer was processed by a dedicated thread-pool with a configurable number of threads. The server was also configured with a limit on the number of enqueued query or insert messages; once the limit was reached, the I/O thread would block until a previous message had been processed.

In each of the tests, a test harness was used to generate load on the system. The test harness was capable of generating three kinds of load:

- **Queries**: Send queries to the server and measure the response time.
- **Sketches**: Send sketches directly to the server, bypassing the daemon.
- **Measurements**: Send individual measurements to the daemon.

These three kinds of load correspond to the three performance tests presented in this chapter.

All of the tests were run on Google Cloud Platform (GCP), using the `n1-standard-64` machine type for the server. At the time of this writing, the `n1-standard-64` machine provided 64 CPU cores and 240 GB memory. In addition, a 500 GB persistent SSD disk was mounted on the server for persisting the RocksDB database. The daemon and test harness processes were hosted on `n1-standard-4` machines with 4 CPU cores

and 15 GB memory. All machines were deployed to the same virtual private cloud (VPC) in the `us-west2` region.

6.2. Performance Tests

This section presents the results of three performance tests on the prototype system. The first measures query response time, the second determines the sustained write throughput of the server, and the third evaluates the system's horizontal scalability across multiple daemons.

6.2.1 Query Response Time Test

In this test, the server was first pre-loaded with a dataset containing series with varying numbers of sketches, as shown in Table 6.1. Each inserted sketch was created from an input stream of randomly generated values, sampled uniformly from the range $[1, 10^4]$. The test harness then repeatedly issued the queries in Table 6.2 to the server, measuring the time between sending the query and receiving the first byte of the response. The server was configured to run 30 query processing threads, and the test harness was constrained to at most 30 queries in-flight at a time.

Metric Name	**Number of sketches**
m1	10^2
m2	10^3
m3	10^4
m4	10^5
n1	10^2
n2	10^3
n3	10^4
n4	10^5
c1 through c10	10^1

Table 6.1: Query metric datasets

ID	Query
Q0	`quantile(fetch("m1"), 0.5)`
Q1	`quantile(fetch("m2"), 0.5)`
Q2	`quantile(fetch("m3"), 0.5)`
Q3	`quantile(fetch("m4"), 0.5)`
Q4	`quantile(coalesce(fetch("m1")), 0.5)`
Q5	`quantile(coalesce(fetch("m2")), 0.5)`
Q6	`quantile(coalesce(fetch("m3")), 0.5)`
Q7	`quantile(coalesce(fetch("m4")), 0.5)`
Q8	`quantile(group("hours", fetch("m1")), 0.5)`
Q9	`quantile(group("hours", fetch("m2")), 0.5)`
Q10	`quantile(group("hours", fetch("m3")), 0.5)`
Q11	`quantile(group("hours", fetch("m4")), 0.5)`
Q12	`quantile(group("days", fetch("m1")), 0.5)`
Q13	`quantile(group("days", fetch("m2")), 0.5)`
Q14	`quantile(group("days", fetch("m3")), 0.5)`
Q15	`quantile(group("days", fetch("m4")), 0.5)`
Q16	`quantile(combine(fetch("m1"), fetch("n1")), 0.5)`
Q17	`quantile(combine(fetch("m2"), fetch("n2")), 0.5)`
Q18	`quantile(combine(fetch("m3"), fetch("n3")), 0.5)`
Q19	`quantile(combine(fetch("m4"), fetch("n4")), 0.5)`
Q20	`quantile(combine(fetch("c1"), fetch("c2"), fetch("c3"), fetch("c4"), fetch("c5"), fetch("c6"), fetch("c7"), fetch("c8"), fetch("c9"), fetch("c10")), 0.5)`

Table 6.2: Queries sent to the server, using the query operators defined in Chapter 5

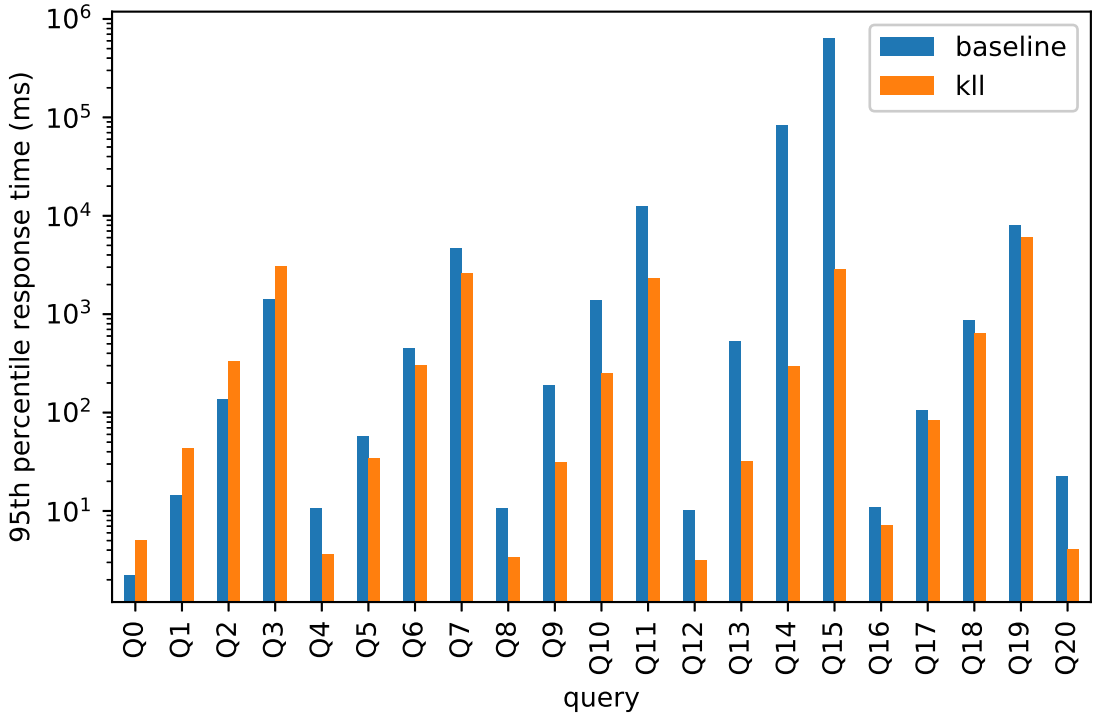

Figure 6.1: Query response times

Figure 6.1 shows the results of the test for both the baseline and KLL sketch implementations. The KLL sketch outperforms the baseline sketch in every query tested, except for Q0 through Q3. Unlike the other queries, Q0 through Q3 do not require any sketches to be merged, so the query pipeline processes relatively small sketches. Both the baseline and KLL implementations need to sort the values stored in each sketch to calculate quantiles. However, whereas the KLL sketch performs a weighted binary search in $O(\log n)$ time to approximate the quantile, the baseline sketch can retrieve the exact quantile in $O(1)$ time. This explains why the baseline sketch outperforms the KLL sketch on queries that do not merge sketches. The other queries, however, do involve merges. Querying a merged baseline sketch is much more expensive, since the merged sketch retains every value from its constituent sketches. In these larger baseline sketches, sorting becomes a bottleneck. In contrast, the merged KLL sketch retains a small subset of the original values. As a result, it needs to sort far fewer values, thus resulting in faster response times.

In addition, these data show a correlation between the number of sketches fetched by the query and the query's response time. The fastest queries tested have 95th percentile response time under 5 ms; the most expensive queries (Q3, Q7, Q11, Q15, and Q19) have 95th percentile response time below 6.5 seconds. Unsurprisingly, retrieving and processing more sketches result in higher response times. The slowest queries each process at least 10^5 sketches, compared to 10^2 sketches for the fastest queries. Even so, these response times are all within an acceptable range for the application monitoring use cases we saw in Chapter 1.

For the KLL sketch implementation, we can compare queries to examine the relative costs of merging sketches and estimating quantiles. At one extreme, Q0 estimates 100 quantiles without performing any merges; at the other extreme, Q4 merges 100 sketches into a single sketch and performs a single quantile estimate.

Interestingly, the response times of these two queries are about the same: 5 ms for Q0 and 3 ms for Q4. This suggests a balance between merging and quantile estimation. It takes time to merge sketches, but the merge process usually discards datapoints, thus resulting in smaller sketches that can estimate quantiles more quickly. For the queries tested, the cost of the merges seems to cancel out the reduction in quantile estimation cost. The pattern holds for queries Q8 and Q12 which combine sketches within the same hour and day, respectively, with approximately the same response times.

Overall, these results show the advantage of KLL sketches over competing systems that store every datapoint. By retaining fewer datapoints, the system can execute queries more efficiently, resulting in faster query response times. Additionally, this test validates that a system based on KLL sketches can respond to user queries in a reasonable amount of time, even when many queries are issued in parallel.

6.2.2 Sketch Insert Stress Test

The sketch insert stress test sent sketches directly to the server, with the goal of measuring maximum sustained throughput. To achieve this, the test harness was configured to establish 50 TCP connections to the server. The harness then repeatedly sent sketches through these connections as quickly as the server could receive them. The server was configured to enqueue at most $65,536$ sketches before blocking. To ensure repeatable results, the server database was emptied before each test run. As in the query response time test, both the baseline and KLL sketch implementations were tested. In addition, the test was repeated with different-sized sketches to see how sketch size affected write throughput. The sketch insert rate was recorded once per minute over an 8 minute interval.

The test results are shown in Figure 6.2. In the graph, we can see that through-

put dropped after the first minute for every variation tested. This was an anomaly caused by the server's request queue; after the first minute, the queue had filled, so new sketches were inserted only as quickly as the server could process them. We see that throughput stabilized after the first minute to the true sustained write throughput.

The data show that KLL sketches allow a higher sustained write throughput than do baseline sketches, regardless of sketch size. In addition, smaller sketches can be inserted more quickly than larger sketches. With fewer bytes to read from the network or write to disk, the server was able to insert the sketch faster.

Using these results, we can estimate the sustained write throughput in terms of the datapoints summarized by the KLL sketches, as shown in Table 6.3. We can see that the large KLL sketches, each representing 10^5 datapoints, had the highest overall throughput. This suggests that the write throughput of the system depends partly on whether datapoints are concentrated in a few sketches, or spread out among many sketches.

Sketch	**Datapoints per sketch**	**Throughput (datapoints / sec)**
Baseline	10^3	1.665429×10^6
Baseline	10^4	5.019512×10^6
Baseline	10^5	2.766356×10^7
KLL	10^3	5.072713×10^7
KLL	10^4	2.378914×10^8
KLL	10^5	2.095069×10^9

Table 6.3: Server sustained write throughput in datapoints per second

Whereas the baseline variation achieved between 1 and 20 million insertions per second, the KLL variation achieved between 50 million and 2 billion insertions per second. By storing a small subset of the values from the original data stream, a system based on KLL sketches can support a much higher sustained write throughput. This compares favorably with existing systems—as noted in Chapter 2, other authors

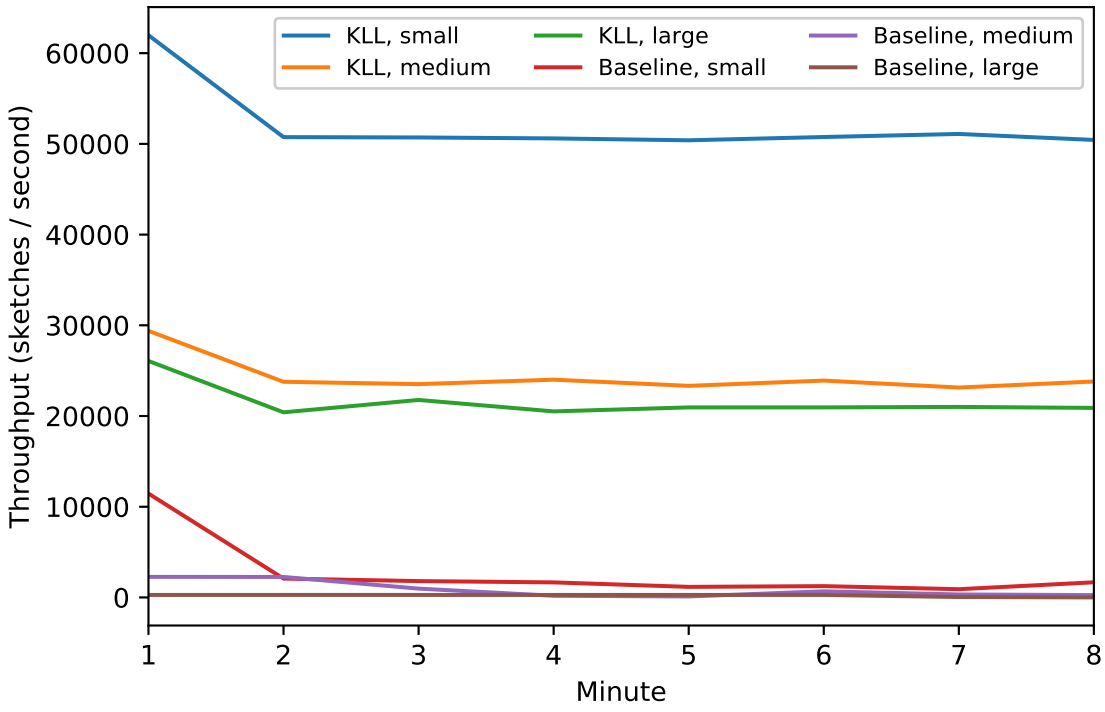

Figure 6.2: Server sustained write throughput in sketches per second. Small sketches represented 10^3 datapoints, medium sketches represented 10^4 datapoints, and large sketches represented 10^5 datapoints

have reported ingestion rates of hundreds of thousands (Dix, 2016) to tens of millions per second (Pelkonen et al., 2015). This represents an improvement of one or two orders of magnitude over the write throughput of competing systems.

6.2.3 Full System Test

Whereas the previous tests evaluated the server in isolation, the full system test measured the interactions between the server and the daemons. Rather than sending sketches directly to the server, as in the previous test, the test harness inserted individual datapoints to daemon instances, as shown in Figure 6.3. In effect, the test harness acted as an instrumented application generating response time measurements. Each daemon then collected those measurements into sketches sent to the server.

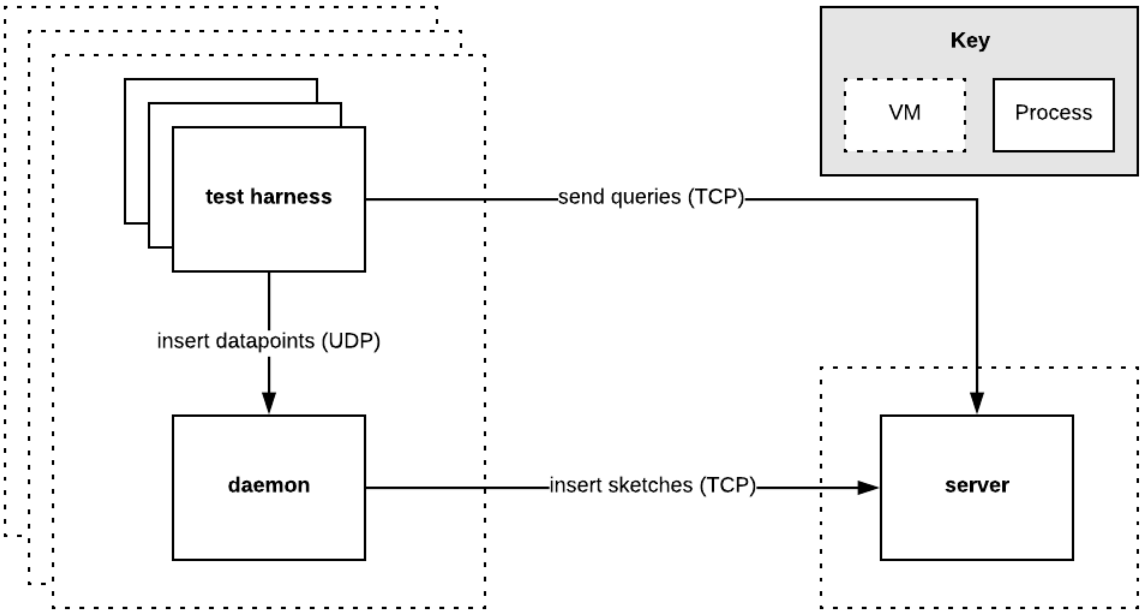

Figure 6.3: Network topology of the full system test.

The full system test used 100 `n1-standard-4` (4-core) machines, each hosting five test harness processes and a single daemon process. The test harness inserted

datapoints for 1000 distinct series, generating measurements uniformly at random from the range $[0, 5000]$. In addition, each test harness process sent user queries directly to the server, with at most 16 queries per process in-flight at once. The server was configured to use 30 write worker threads and 30 read worker threads, with downsampling performed in a separate background thread every 10 minutes. As in the previous tests, both the baseline and KLL sketch implementations were evaluated.

Each daemon instance achieved a sustained write throughput of approximately $150,000$ datapoints per second, with stable memory and CPU usage. The limiting factor appeared to be the number of UDP packets the daemon could process per second, because the total throughput remained constant even after increasing the number of test harness processes on a machine with 32 CPU cores. Across all 100 daemons, the total write throughput of the system was 15 million datapoints per second.

Surprisingly, both the baseline and KLL sketch implementations were able to support the same sustained write throughput. This shows that both the baseline and KLL sketch implementations benefited from horizontally scaling sketch construction across daemon instances. Unlike in the sketch insert stress test, the server was able to "keep up" with the sketches inserted by the 100 daemon instances. As a result, neither implementation was limited by the server's capacity to ingest sketches.

However, the CPU, memory, and network usage of the daemon and server show the advantages of the KLL sketch implementation over the baseline. Figure 6.4 shows that the KLL sketch implementation required both less memory and less CPU to construct. Since the KLL sketch retains fewer datapoints, it uses less memory. Likewise, since datapoints must be sorted when encoding the sketch to Stream VByte, fewer datapoints results in faster encoding, which in turn results in more efficient

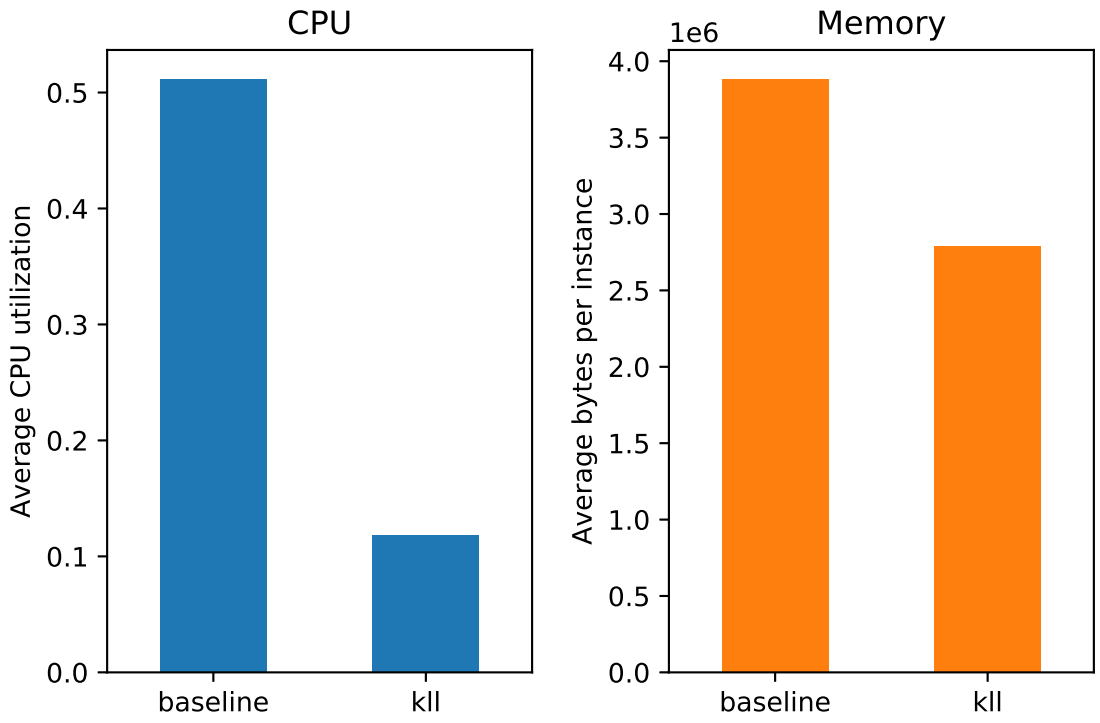

Figure 6.4: Daemon CPU and memory usage

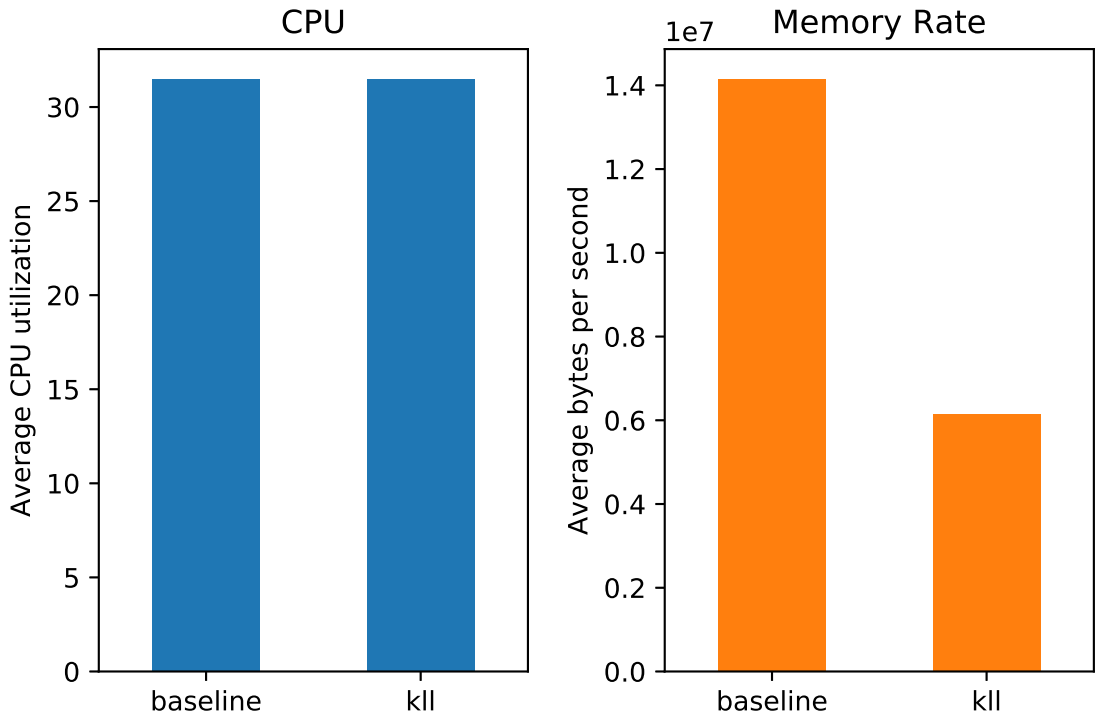

Figure 6.5: Server CPU usage and memory rate (increase in memory usage over time)

CPU utilization. On the server side (Figure 6.5), the two sketch implementations had similar CPU usage, but the KLL sketch used less memory.[1]

The full system test validates that the daemons and server can work in concert to achieve high write throughput. If the system had been overloaded, the daemon would have been unable to flush sketches to the server quickly enough to "keep up" with incoming datapoints. In that scenario, one would expect memory usage on the daemon to increase over time. The fact that the daemon's memory usage remained constant during the full system test shows that the system can support high sustained write throughput. Furthermore, the test shows the benefits of horizontally scaling inserts across multiple daemon instances. While each individual daemon can process a limited number of UDP packets per second, collectively, they can achieve a much higher write throughput.

6.3. Conclusion

Taken together, the three performance tests demonstrate the advantages of using KLL sketches in an application monitoring system. By storing fewer datapoints, KLL sketches allowed the system to respond to user queries more quickly. Likewise, by strategically discarding datapoints, the KLL sketch implementation dramatically reduced the number of datapoints sent from the daemons to the server, resulting in at least an order-of-magnitude in write throughput.

The next chapter places these results in context and suggests directions for future research.

[1]The figure shows the memory *rate* (increase in bytes per second) rather than overall memory usage. This is because during the test RocksDB could store the entire dataset in memory, so memory usage increased linearly throughout the test.

Chapter 7: Conclusion

This chapter returns to the question posed at the beginning of the thesis: "Could mergeable quantile sketches provide the basis of an application monitoring system?" The experimental results presented in Chapter 6 provide evidence in favor of this approach, but they also suggest directions for further investigation.

Relative to the competing systems surveyed in Chapter 2, we saw that a system based on quantile sketches offers several advantages. First, quantile sketches greatly reduce the amount of data transmitted, stored, and retrieved by the system. Second, merging quantile sketches allows the system to downsample data to further reduce space, with strong theoretical bounds on quantile approximation error. Finally, quantile sketches can be merged at query-time to answer aggregate queries across multiple series or time windows.

However, the experiments in this thesis used only synthetic data and user queries. While these can show the range of behaviors the application monitoring system might exhibit, they cannot prove conclusively how the system would perform when monitoring real services. For example, we saw in Chapter 3 that Stream VByte could achieve greater compression ratios on datasets with small deltas between sorted values. Although it is plausible that response time measurements would be tightly clustered, one would need to validate this assumption using real-world datasets to be sure. Likewise, Chapter 6 showed that queries involving many sketch merges have relatively high response times, but it is unclear how prevalent such queries are in real-world application monitoring systems. It would be valuable to evaluate the system

using query patterns similar to those used by operations teams in the field. Finally, the full-scale tests used a fixed sketch size of $k = 200$ (where k is the capacity of the top-level compactor); it may be possible to reduce the sketch size further if operations teams can tolerate the error this would introduce.

The results in this thesis suggest several directions for future research. One line of inquiry could introduce parallelism to improve query performance. The prototype system executes each query pipeline serially, merging one sketch at a time. However, modern database systems such as MonetDB (Boncz et al., 2005) typically parallelize query operations to reduce latency. One can imagine a system that parallelizes sketch merges in order to improve read performance. Since parallelism generally introduces some coordination overhead, it would be productive to investigate which queries would benefit from parallelism.

Another potential research topic is clustered storage. The prototype system wrote all received data to a single RocksDB key-value database. As suggested in Chapter 4, it would be possible to replace the RocksDB database with Cassandra (Lakshman & Malik, 2010) or DynamoDB (DeCandia et al., 2007) to provide high availability and horizontal scaling of storage capacity. This would also make it possible to horizontally scale the number of backend instances that insert data and process queries. One might predict that using a Cassandra or DynamoDB as the storage engine would introduce additional latency when inserting or retrieving sketches. Future research could evaluate the impact of using clustered storage on the system's performance.

Finally, the prototype system inherited a restrictive data model from Graphite (Davis, 2011) in which each measurement is associated with a single series. As we saw in Chapter 4, this simplified the interactions between the backend instance and the key-value store. However, this restriction can be inconvenient for application

developers, since they must encode every attribute of a measurement in its series string. Systems such as InfluxDB (Dix, 2017) support a more flexible data model in which each measurement can be assigned one or more "tags." Users can then fetch the relevant measurements by specifying which tags should be included or excluded from the query. It would be fruitful to explore how this feature would impact the structure and performance of the query subsystem.

In conclusion, I believe the results from this thesis make a plausible case for using mergeable quantile sketches in application monitoring systems. The last several decades have seen a tremendous increase in the scale and complexity of distributed systems. High-quality application monitoring has become critical to running network services in production. If these trends continue, as I anticipate they will, application monitoring systems will need to store and query a growing volume of data. From this perspective, mergeable quantile sketches provide a promising starting point for the next generation of application monitoring systems.

References

Abadi, D., Madden, S., & Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (pp. 671–682).: ACM.

Agarwal, P. K., Cormode, G., Huang, Z., Phillips, J. M., Wei, Z., & Yi, K. (2013). Mergeable summaries. *ACM Transactions on Database Systems (TODS)*, 38(4), 26.

Boncz, P. A., Zukowski, M., & Nes, N. (2005). MonetDB/X100: Hyper-pipelining query execution. In *Cidr*, volume 5 (pp. 225–237).

Buragohain, C. & Suri, S. (2009). Quantiles on streams. In *Encyclopedia of Database Systems* (pp. 2235–2240). Springer.

Cormode, G. (2009). Count-min sketch. In *Encyclopedia of Database Systems* (pp. 511–516). Springer.

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

Davis, C. (2011). Graphite. In A. Brown & G. Wilson (Eds.), *The Architecture of Open Source Applications*.

Dayan, N. & Idreos, S. (2018). Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging. In *Proceed-*

ings of the 2018 International Conference on Management of Data (pp. 505–520).: ACM.

Dean, J. & Barroso, L. A. (2013). The tail at scale. *Communications of the ACM*, 56(2), 74–80.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41 (pp. 205–220).: ACM.

Dix, P. (2016). InfluxDB 0.10 release supports more writes per second. Retrieved 2018-02-15 from https://www.influxdata.com/blog/announcing-influxdb-v0-10-100000s-writes-per-second-better-compression/.

Dix, P. (2017). InfluxDB Storage Internals. Retrieved 2018-02-15 from https://www.youtube.com/watch?v=rtEalnKT25I.

Dunning, T. & Ertl, O. (2019). Computing extremely accurate quantiles using t-digests. Retrieved from https://arxiv.org/abs/1902.04023.

Etsy (2016). StatsD documentation. Retrieved 2018-02-22 from https://github.com/etsy/statsd/tree/master/docs.

Facebook Engineering Team (2017). RocksDB basics. Retrieved 2018-03-17 from https://github.com/facebook/rocksdb/wiki/RocksDB-Basics.

Felber, D. & Ostrovsky, R. (2015). A randomized online quantile summary in o (1/epsilon* log (1/epsilon)) words. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 40: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Gan, E., Ding, J., Tai, K. S., Sharan, V., & Bailis, P. (2018). Moment-based quantile sketches for efficient high cardinality aggregation queries. *Proceedings of the VLDB Endowment*, 11(11).

Grafana Labs (2018). Grafana documentation. Retrieved 2018-12-09 from http://docs.grafana.org/.

Greenwald, M. & Khanna, S. (2001). Space-efficient online computation of quantile summaries. In *ACM SIGMOD Record*, volume 30 (pp. 58–66).: ACM.

Hoare, C. A. (1961). Algorithm 65: find. *Communications of the ACM*, 4(7), 321–322.

InfluxData Inc (2018). InfluxDB 1.7 documentation. Retrieved 2018-12-11 from https://docs.influxdata.com/influxdb/v1.7/.

Jensen, S. K., Pedersen, T. B., & Thomsen, C. (2017). Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 29(11), 2581–2600.

KairosDB Team (2015). KairosDB documentation v1.2.0. Retrieved 2018-12-22 from https://kairosdb.github.io/docs/build/html/index.html.

Karnin, Z., Lang, K., & Liberty, E. (2016). Optimal quantile approximation in streams. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on* (pp. 71–78).: IEEE.

Kuszmaul, B. (2010). *How TokuDB fractal tree indexes work*. Technical report, Technical report, TokuTek.

Lakshman, A. & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2), 35–40.

Lang, E. (2016). kll.py. Retrieved 2019-01-09 from https://github.com/edoliberty/streaming-quantiles/blob/8aff95620304ab9db72058280af8107d5daa819f/kll.py.

Lemire, D., Kurz, N., & Rupp, C. (2018). Stream vbyte: Faster byte-oriented integer compression. *Information Processing Letters*, 130, 1–6.

Malpas, I. (2011). Measure anything, measure everything. Retrieved 2018-02-22 from https://codeascraft.com/2011/02/15/measure-anything-measure-everything/.

Manku, G. S., Rajagopalan, S., & Lindsay, B. G. (1999). Random sampling techniques for space efficient online computation of order statistics of large datasets. In *ACM SIGMOD Record*, volume 28 (pp. 251–262).: ACM.

Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., et al. (2013). Scaling memcache at facebook. In *nsdi*, volume 13 (pp. 385–398).

O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. (1996). The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4), 351–385.

Pelkonen, T., Franklin, S., Teller, J., Cavallaro, P., Huang, Q., Meza, J., & Veeraraghavan, K. (2015). Gorilla: A fast, scalable, in-memory time series database. *Proceedings of the VLDB Endowment*, 8(12), 1816–1827.

Pibiri, G. E. & Venturini, R. (2018). Inverted index compression. *Encyclopedia of Big Data Technologies*, (pp. 1–8).

Sathe, S., Papaioannou, T. G., Jeung, H., & Aberer, K. (2013). A survey of model-based sensor data acquisition and management. In *Managing and mining sensor data* (pp. 9–50). Springer.

Seltzer, M. & Bostic, K. (2011). BerkeleyDB. In A. Brown & G. Wilson (Eds.), *The Architecture of Open Source Applications*.

Shrivastava, N., Buragohain, C., Agrawal, D., & Suri, S. (2004). Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems* (pp. 239–249).: ACM.

Teller, J. (2018). Beringei: A high-performance time series storage engine. Retrieved 2018-03-04 from https://code.fb.com/core-data/beringei-a-high-performance-time-series-storage-engine/.

The OpenTSDB Authors (2010). OpenTSDB - A Distributed, Scalable Monitoring System. Retrieved 2017-08-06 from http://opentsdb.net/overview.html.

The Rust Project Developers (2018). Bencher 0.1.5. Retrieved 2019-02-04 from https://github.com/bluss/bencher/releases/tag/0.1.5.

Uber Engineering (2018). M3: Uber's open source, large-scale metrics platform for Prometheus. Retrieved 2018-12-22 from https://eng.uber.com/m3/.

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

Wang, C. (2015). Statsd-proxy. Retrieved 2018-12-22 from https://github.com/hit9/statsd-proxy.

Wang, L., Luo, G., Yi, K., & Cormode, G. (2013). Quantiles over data streams: an experimental study. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (pp. 737–748).: ACM.

Yahoo! Inc. (2018). KllFloatsSketch.java v0.12.0. Retrieved 2019-01-09 from https://github.com/DataSketches/sketches-core/blob/master/src/main/java/com/yahoo/sketches/kll/KllFloatsSketch.java.

Appendix A: KLL Sketch Insert Time Complexity

This chapter establishes an upper bound on the time to insert a value into a KLL sketch.

Lemma A.1. *Each of n values inserted into a KLL sketch with top-level capacity k is compacted at most $O(\log k)$ times.*

Proof. Let C denote the maximum number of times any of the n inserted values is compacted. At each compaction, a value is either discarded or preserved. If it is discarded, then it can participate in no more compactions. Otherwise, if it is preserved, it is inserted into the next compactor in the hierarchy. Therefore, C is at most the number of compactors in the hierarchy after n insertions.

The number of compactors in the hierarchy includes only compactors with capacity greater than 2, because once a compactor's capacity reaches 2 (the minimum possible), it is absorbed into the sampler. By definition of a KLL sketch, (a) the top $O\big(\log log \frac{1}{\delta}\big)$ compactors have capacity k, where $k > 2$, and (b) the remaining compactors at depth d have capacity $k\left(\frac{2}{3}\right)^d$. The lowest-level compactor, at depth C, must therefore have capacity greater than 2, so $k\left(\frac{2}{3}\right)^C > 2$. We can then solve for C:

$$\log\left[k\left(\frac{2}{3}\right)^C\right] > \log 2 \tag{A.1}$$

$$\log k - C\log\frac{3}{2} > \log 2 \tag{A.2}$$

$$C < \frac{\log k - \log 2}{\log\frac{3}{2}} \tag{A.3}$$

which implies that C is $O(\log k)$.

□

Theorem A.1. *A KLL sketch with top-level capacity k can insert n values in $O(\log k)$ amortized time per insert.*

Proof. Let s denote the number of items in a compactor. The cost of compaction at the lowest level is dominated by the time to sort the s values, which is $O(s \log s)$ on average. Chapter 3, Section 3.3 showed that the other compactors need only to merge already-sorted sequences, so compaction cost at subsequent levels is only $O(s)$. Note that because compactor capacities decay exponentially from k, and a compactor cannot exceed its capacity, each compactor has $s \leq k$.

When compaction occurs at the lowest level, use $O(\log s)$ from each of the s values stored in that compactor to cover the $O(s \log s)$ cost of the sort. For compactions at each of the other levels, use $O(1)$ for each of the s values to cover the $O(s)$ cost of merging sorted sequences. From Lemma A.1, there are at most $C = O(\log k)$ compactors. Furthermore, since $s \leq k$ the total cost of all compactions for each value is $O(\log k + (C - 1)) = O(\log k)$. Assuming that the time to insert into the random sampler is $O(1)$, this yields an amortized cost per insert of $O(\log k)$. □